PL/JSON Reference Guide (version 1.0.5)

For Oracle 10g and 11g

Jonas Krogsbøll

Contents

1 PURPOSE

The goal of PL/JSON is to create a correct implementation of JSON to use in a PL/SQL environment. The Oracle object syntax has been choosen to ensure a straightforward and easy way to decode and encode JSON. PL/JSON is delivered AS IS and we cannot make any guarantee or be held responsible to any unwanted effects that may arise from using this software. Although we would like to stress that we have tested and used this software and would like to think that is is a safe product to use.

2 DESCRIPTION

Parsing and emitting JSON is done in two packages, but you will rarely need to access these directly. The essential objects are linked to the relevant functions in the packages (in constructors and output methods). Basically PL/JSON can be used in two ways: Either you manually build up an object structure and emit JSON text with the to_char method or you parse JSON text into a object structure and use the objects in PL/SQL. Obvious you could also parse JSON text into objects, modify these and then emit JSON text. There are only three objects you should know: JSON, JSON_LIST and JSON_VALUE. The JSON object can hold an object described by the { } syntax and is named JSON rather than JSON_OBJECT to keep the name short and the fact that an object with the name object sounds silly. The JSON_LIST object can hold an array described with the [ ] syntax. The postfix "list" was choosen over "array" for two reasons, one: to keep it short, two: there seams to be a naming standard in oracle types that the postfix "array" is being used to describe types with the "table of" construction. The last type JSON_VALUE contains the primitive simple types (strings, numbers, bools, null), but can also contain an array or an object. The object model for PL/JSON is shown on figure 1.

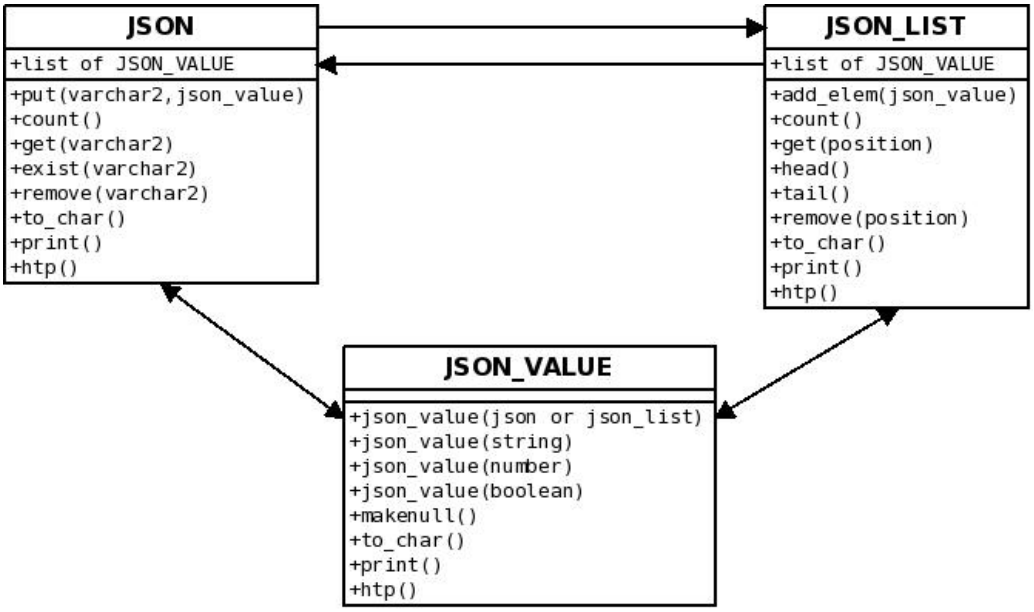


Figure 1: Essential Objects in PL/JSON

3 IN THE RELEASE

- Install script
- Uninstall script.
- 4 new oracle types ready to use in your database.
- 3 packages (parser, printer and extension)
- A few examples files
- Some testing scripts - creates and delete a table (JSON_TESTSUITE)
- Optional addons packages.

4 GETTING STARTED

To get started using this product, you should first install the product with the install script and then take a look at the examples in the *examples* folder. The content of each example file are:

- **ex1.sql**: Simple creation of a JSON object.
- **ex2.sql**: Simple creation of a JSON list.
- **ex3.sql**: Working with parser exceptions.
- **ex4.sql**: Building an object with the API.
- **ex5.sql**: Building a list with the API.
- **ex6.sql**: Working with variables as copies.
- **ex7.sql**: Using the extension package.
- **ex8.sql**: Using JSON Path getters.
- **ex9.sql**: Using JSON Path putters.
- **ex10.sql**: Using JSON Path remove.
- **ex11.sql**: Using the TO_CLOB method to save JSON.
- **ex12.sql**: Pretty print with JSON Path.
- **ex13.sql and ex14.sql**: Binary support with base64.
- **ex15.sql**: Conversion between JSON and JSON_LIST.
- **ex16.sql**: Dynamic JSON (requires installation of the json_dyn package).
- **ex17.sql**: Duplicate check and fast creation of JSON.
- **ex18.sql**: Convert XML to JSON_LIST with JSONML.
- **ex19.sql**: Output unescaped strings.

You can also ask questions in the support forum (http://sourceforge.net/projects/pljson/forums/forum/935365) or search the internet for information - google is your friend!

5 TWEAKS

You can tweak the behaviour of PL/JSON by setting various variables in the packages. Here comes a description of the possible adjustments you can do.

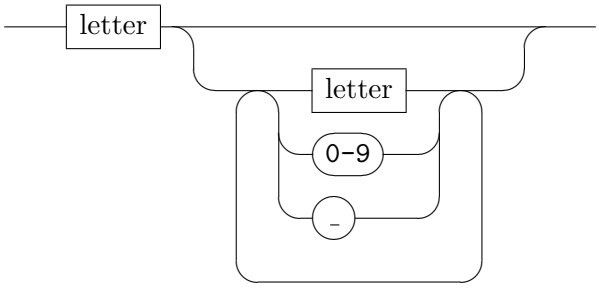
JSON_PRINTER

In the printer package you can edit the line break method to linux, windows or mac style. You can also change the indentation to the tabular char, or change the number of spaces.

JSON_PARSER

The parser is extended to accept more than just pure JSON. The variable "json_strict" is defaulted to false and can be set to true to force PL/JSON to only accept proper JSON. The current implementation allows input to contain comments with the /* */ notation. Furthermore it extends the definition of a string to allow singlequotes and converts names into strings. The railroad diagram below shows the grammar for names accepted by PL/JSON:

name



However, the values "true", "false" and "null" will not be handled as names.
Example of usage with the extended grammar:

```
{
  abc : '123"'
}
```

Parsed and emitted:

```
{
  "abc" : "123\""
}
```

JSON_PARSER: Javascript functions

Even though javascript functions are not a part of JSON, you might want to emit functions if your receiving client is a browser. This is how you do it:

```
declare
  obj json := json();
begin
  obj.put('test', json_value('function() {return 1;}', esc => false ));
  obj.print;
end;
```

The output is:

```
{
  "test": /**/function() {return 1;}/**/
}
```

Which is not valid JSON but will work in javascript.
The surrounding /**/ is a message to the parser to start and stop building a unescaped json_value string:

```
declare
  obj json := json('{"test": /**/function() {return 1;}/**/}');
begin
  obj.print;
end;
----
{
  "test": /**/function() {return 1;}/**/
}
```

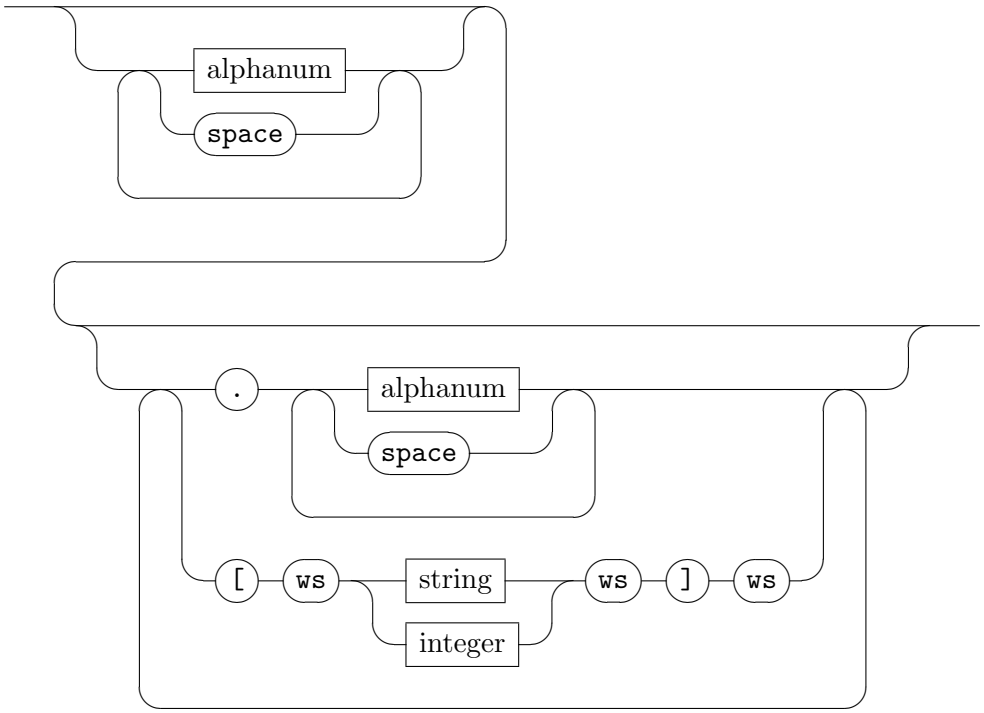
JSON_EXT

The extension package is now mandatory. It contains the path implementation and adds support for dates and binary lob's. Dates is not a part of the JSON standard, so it's up to you to specify how you would like to handle dates. The current implementation specifies a date to be a string that which follows the format: yyyy-mm-dd hh24:mi:ss. If your needs differ from this, then you can rewrite the functions in the package.

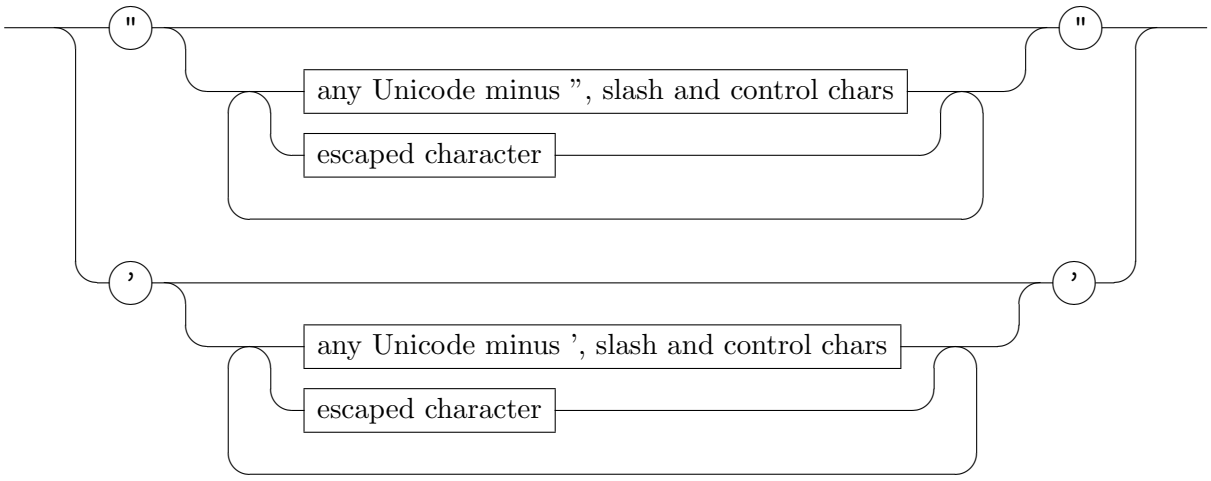
6 JSON PATH

A JSON Path is a way to navigate in a JSON object. The implementation is quite simple and does not support all the features that are available in stefan goessners JavaScript, Python and PHP implementation of JSON Path. Actually, all that is does is to add the support for navigation in JSON, that are allready build in into those languages. When navigating with JSON Path in PL/JSON, members of JSON objects are found with the dot operator while elements in lists are found with square brackets. Accepted input in path navigation is formalized with these railroad diagrams (ws is insignificant whitespace):

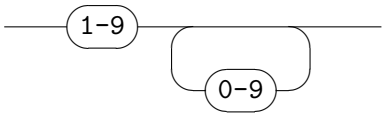
jsonpath



string



integer



From version 0.9.6 the implementation accepts an extended grammar where you use a zero-indexed JSON Path.
The following examples show how you can use JSON Path to extract from a JSON object.

The JSON Object:

```
{
  "xyz" : {
    "abc" : [1,2,3,[4,5,{"123":45}]]
  }
}
```

Extract the "abc" list:

```
json_ext.get_json_list(obj,
  'xyz.abc').print;
-----------
[1, 2, 3, [4 ,5 , {
  "123" : 45
}]]
```

Extract the "123" number:

```
json_ext.get_number(obj,
  'xyz.abc[4][3].123').print;
----------
45
```

As of version 0.8.4, square brackets can be used to extract JSON members like you would do in JavaScript:

```
json_ext.get_number(obj,
  '["xyz"]["abc"][4][3]["123"]').print;
----------
45
```

You can also use JSON Path to modify an existing JSON Object:

```
json_ext.put(obj, 'xyz.abc',
  json('{"val":123}'));
----------
{
  "xyz" : {
    "abc" : {
      "val" : 123
    }
  }
}
```

Remove unwanted elements is also an option:

```
json_ext.remove(obj, 'xyz.abc');
----------
{
  "xyz" : {
  }
}
```

In the 0.9.1 release, both JSON and JSON_LIST are hooked to the JSON Path implementation:

```
declare
  v_obj json := json('{a:true}');
  v_list json_list := json_list('[1,2,[3,4]]');
begin
  v_obj.path('a').print;
  v_list.path('[3][1]').print;
end;
----------
true
3
```

The example files 8 to 10 provides a more detailed explaination.

In the 0.9.2 release, it is also possible to use path to modify objects and arrays:

```
declare
  v_obj json := json('{a:true}');
  v_list json_list := json_list('[1,2,[3,4]]');
begin
  v_obj.path_put('a[2]', json_list('[true, true]'));
  v_obj.print;
  v_list.path_put('[3][1]', 'test');
  v_list.print;
end;
----------
{
  "a" : [null, [true, true]]
}
[1, 2, ["test", 4]]
```

7 BEHAVIOR & ERROR HANDLING

Input to the parser is expected to be in the charset of the database. The objects that are generated contains unescaped values that will be escaped when emitted through the printer. To ensure correct JSON output, even from non-UTF databases, only ASCII chars are emitted. All the characters which are not part of ASCII will be escaped.
The errors or exceptions that PL/JSON may throw, can be catched with the following code:

```
declare
  scanner_exception exception;
  pragma exception_init(scanner_exception, -20100);
```

```
  parser_exception exception;
  pragma exception_init(parser_exception, -20101);
  jext_exception exception;
  pragma exception_init(jext_exception, -20110);
  ...
begin
  ... json code ...
exception
  when scanner_exception exception then ...
  when parser_exception exception then ...
  when jext_exception exception then ...
end;
```

8 KNOWN LIMITATIONS

- key-names are limited to 4000 characters.
- The number parsing assumes that oracles number type can contain the input (in most cases it can).

9 TESTSUITE

Any proper product is tested for correctness. So should PL/JSON be, with a testsuite that can be executed without installing any additional software on your database. You probaly don't need the testsuite, but if you modify the implementation or add more features, tests will be needed. Also if you discover a bug, you could report the bug by writing a relevant testcase.

10 CONTRIBUTING

Write to us in the forums of sourceforge.net. We will be happy to hear from you.

Q: "I've added a lot of code, please merge my changes"
A: Hmmm - it's not that we don't appriciate your work, but we would really prefer that you wrote tests and documentation to each feature - otherwise new code could easily break functionality.

Q: "I've added some changes and I might contribute them to the project, but what's in it for me?"
A: This is not GPL, so you can keep your changes if you want to. When you are contributing then more eyes will look at your code. Possible errors might get detected and corrected and new features may arrise from your features - making it a better product you can use.

11 OPTIONAL PACKAGES

- JSON_DYN A package that enables you to generate JSON from sql. Nested queries are not supported. See example 16 for more information.
- JSON_ML A package that converts from XML to JSON using a XSLT stylesheet. See www.jsonml.org.
- JSON_XML A package that converts a JSON object to XML.
- JSON_HELPER Work on JSON with set operations.

- JSON_UTIL_PKG Written by Morten Braten (http://ora-00001.blogspot.com). Generate JSON from sql using a XSLT stylesheet.
- JSON_AC Autocomplete package. Some PL/SQL IDE's provide autocompletion when using a package but not when using an object type. This package is a wrapper around the methods on JSON, JSON_LIST and JSON_VALUE. Use it if you often forget the methods on those object types.