Using tDOM to work with JSON data

Rolf Ade

rolf@pointsman.de

Status

- Follows RFC 8259, parses any valid JSON (fully true with Tcl 9/tDOM 0.9.4)
- Preserves all JSON datatypes (including the symbol names true, false, null)
- Parsing and serializing is a full round, no information will be lost (other than the formatting)
- This includes even duplicated object member names (they are not forbidden) and the order of the members of an object
- It is surely much faster than any script approach
- There are easy patterns to process the data
- There is a nice tclish pattern to create JSON data

JSON is just nested data

```
{
     "Image": {
         "Width":  800,
         "Height": 600,
         "Title":  "View from 15th Floor",
         "Thumbnail": {
            "Url":    "http://www.example.com/image/481989943",
            "Height": 125,
            "Width":  100
         },
         "Animated" : false,
         "IDs": [116, 943, 234, 38793]
     }
}
```

This is **not** JSON to XML conversion

- Such a conversion is rarely needed or useful
- If you have XML, then parse without `-json` and serialize with `dom asXML`
- If you have JSON, then parse with `-json` and serialize with `dom asJSON`
- `$doc asJSON` on a random XML DOM tree is most probably sense- and useless
- $doc asXML on a JSON DOM tree helps to illustrate how the DOM tree looks like and how to build XPath expressions to extract the JSON data

Parse JSON with tDOM

- Add `-json` as option to [`dom parse`]

  ```
  set doc [dom parse -json $jsonstring]
  ```

- Atm only parsing of strings, the `-channel` option can't be used with `-json`
- Use Tcl 9 to prevent that encoding errors slip through
- Other JSON related `[dom parse]` options:

  ```
  -jsonroot <docelmname>              (we come to this in a moment)
  -jsonmaxnesting <integer>           (default 2000)
  --                                  (end of options, -1.23 is valid
                                      JSON)
  ```

JSON is a forest

```
set doc [dom parse -json {{
        "precision": "zip",
        "Latitude":  37.371991,
        "Longitude": -122.026020,
        "Address":   "",
        "City":      "SUNNYVALE",
        "State":     "CA",
        "Zip":       "94085",
        "Country":   "US"
     }
}]
```

```
$doc asXML  ;# =>
<precision>zip</precision>
<Latitude>37.371991</Latitude>
<Longitude>-122.026020</Longitude>
<Address></Address>
<City>SUNNYVALE</City>
<State>CA</State>
<Zip>94085</Zip>
<Country>US</Country>
```

- The document node has the type OBJECT (`$doc jsonType => OBJECT`)
- The tDOM DOM model handles this just fine
- Calls as `$doc childnodes` or `$doc selectNodes` work as expected
- `$doc documentElement` isn’t useful for such DOM “forests”

JSON is a forest II

```
set doc [dom parse -json -jsonroot myroot {{
        "precision": "zip",
        "Latitude":  37.371991,
        "Longitude": -122.026020,
        "Address":   "",
        "City":      "SUNNYVALE",
        "State":     "CA",
        "Zip":       "94085",
        "Country":   "US"
      }
}]
```

```
$doc asXML -indent 4; # =>
<myroot>  (jsonType OBJECT)
    <precision>zip</precision>
    <Latitude>37.371991</Latitude>
    <Longitude>-122.026020</Longitude>
    <Address></Address>
    <City>SUNNYVALE</City>
    <State>CA</State>
    <Zip>94085</Zip>
    <Country>US</Country>
</myroot>
```

- Use the -jsonroot option if you prefer the JSON data to be under a single root element (mental model as with XML)
- Then the pattern is:

```
set doc [dom parse -json -jsonroot myroot $jsondata]
set jsonroot [$doc documentElement]
```

JSON is a forest III

```
set doc [dom parse -json -jsonroot myroot {
    [
        {
          "precision": "zip",
          "Latitude":  37.7668,
          "Longitude": -122.3959
        },
        {
          "precision": "zip",
          "Latitude":  37.371991,
          "Longitude": -122.026020
        },
        ["a","b","c"]
    ]
}]
```

```
$doc asXML -indent 4; # =>
<myroot>  (jsonType ARRAY)
    <objectcontainer>
        <precision>zip</precision>
        <Latitude>37.7668</Latitude>
        <Longitude>-122.3959</Longitude>
    </objectcontainer>
    <objectcontainer>
        <precision>zip</precision>
        <Latitude>37.371991</Latitude>
        <Longitude>-122.026020</Longitude>
    </objectcontainer>
    <arraycontainer>abc</arraycontainer>
</myroot>
```

- Note the inserted container elements

JSON is a forest IV

- Object container elements (`objectcontainer`) and array container elements (`arraycontainer`) are inserted during parsing to group the object members and array elements
- Think about the name `objectcontainer` as “{“ and the name `arraycontainer` as “[“
- Could have used that “{“ and “[“ as container element names, but that would have made XPath expressions slightly more convoluted
- There is no conflict between object element members and the names above
- If you really dislike the names you can change them at build-time with -D defines

JSON is typed

- JSON has the structure types object and array and the data types string, number, true, false and null
- For JSON, the string “1.23” is something else than the number 1.23
- The JSON datatype handling is done with an additional property of the DOM nodes and documents
- Inspect or set this property with: `$node jsonType ?jsonType?`
- You can post-validate your DOM tree after parsing
- The tDOM schema language includes text constrains for all basic JSON types

Validation Example

```
tdom::schema s
s defelement JSON {
    element astring {text {jsontype STRING}}
    element anumber {text {jsontype NUMBER}}
    element atrue {text {jsontype TRUE}}
    element afalse {text {jsontype FALSE}}
    element anull {text {jsontype NULL}}
}
set result ""
foreach json {
    {{
        "astring": "0.123",
        "anumber": 0.123,
        "atrue": true,
        "afalse": false,
        "anull": null
    }}
    {{
        "astring": "0.123",
        "anumber": "0.123",
        "atrue": true,
        "afalse": false,
        "anull": null
    }}
} {
    set jdoc [dom parse -json -jsonroot JSON $json]
    lappend result [s domvalidate [$jdoc documentElement]]
    $jdoc delete
}
s delete
set result

=> 1 0
```

Using the data

- After parsing, the resulting $doc is an ordinary DOM tree (just enriched with additional JSON type information)
- You can navigate and access data as usual with tDOM and DOM trees built from XML
- Especially you can use the selectNodes method (XPath queries)

XPath features for JSON

- XPath expressions expect XML names in element steps
- JSON allows any kind of wacky object member names
- At every place in an Xpath expression where an element name is expected, tDOM allows the syntax %tclvarname

  ```
  set member “wacky object member name”
  set thismember [$objectnode selectnodes %member]
  ```

- Inside a tDOM XPath expression, wherever the syntax allows a string, $var can be used and wherever an element name is allowed, %var can be used. Both referring to the Tcl variable “var” in scope
- laststring() function

Just an Example

```
{
  "codes": [
    {
      "alpha_3": "aav",
      "name": "Austro-Asiatic languages"
    },
    {
      "alpha_3": "afa",
      "name": "Afro-Asiatic languages"
    },
    {
      "alpha_3": "alg",
      "name": "Algonquian languages"
    },
    {
      "alpha_3": "apa",
      "name": "Apache languages"
    }
  ]
}
```

```
set doc [dom parse -json $jsondata]
foreach codedata [$doc selectNodes codes/objectcontainer] {
    set code [$codedata selectNodes string(alpha_3)]
    set name [$codedata selectNodes string(name)]
    puts "$name: $code"
}

==>

Austro-Asiatic languages: aav
Afro-Asiatic languages: afa
Algonquian languages: alg
Apache languages: apa
```

Serialize JSON with tDOM

- Use the doc method asJSON to serialize a JSON DOM tree :

```
set jsonstring [$doc asJSON]
```

- Current options of the asJSON method:

```
-indent <”none”,0..8>          (kind of pretty printing)

-channel <channelname>         (write serialization directly to channel)
```

Creating JSON with tDOM

- For simple cases it is tempting to use string commands to create JSON. As for example:

  ```
  subst -nocommands {{"criteria":[{"field":"attributeType","param":"$csname"}]}}
  ```

- This suffers from the injection problem: What, if the value of `csname` is `foo"bar ?`
- If you create JSON with string commands, you have to do escaping on your own
- If you create JSON with tDOM it does the escaping for you automatically while serializing.

Creating JSON with tDOM

- It is possible to create the JSON DOM from scratch with basic DOM methods:

```
set doc [dom createDocumentNode root]
set root [$doc documentElement]
set node_Number  $doc createElement "aNumber"]
set node_Number_Value [$doc createTextNode 0.123]
$node_Number_Value jsonType NUMBER
$node_Number appendChild $node_Number_Value
$root appendChild $node_Number
# etc ...
```

- That tend to get tedious and convoluted
- You probably will need:

```
dom setNameCheck 0
dom setTextCheck 0
```

Creating JSON – the better way

- Use the appendFromScript method
- Create your node creation commands (`dom createNodeCmd`) with -jsonType
- This automatically disables the by default on name and text checks (for this case) without fiddling with the global knobs
- The `[dom createNodeCmd -tagName]` option allows you to create your node creation commands with name prefix (or in a Tcl namespace)
- Create your JSON like it should look like at the end.

How does it look

```
# Create once the “vocabulary” for you JSON target format

# Simple Object members
dom createNodeCmd -jsonType NONE elementNode aNumber
dom createNodeCmd -jsonType NONE elementNode aNumberAsString
dom createNodeCmd -jsonType NONE elementNode aString
dom createNodeCmd -jsonType NONE elementNode "a key with spaces"

# An Array
dom createNodeCmd -jsonType ARRAY elementNode "aArray"

# Data types
dom createNodeCmd -jsonType NUMBER textNode    jsonNumber
dom createNodeCmd -jsonType STRING textNode    jsonString
```

How does it look II

```
# Then create that commands as often as you need to
# create JSON string.

# Create the document node
set resultJSON [dom createDocumentNode]
$resultJSON appendFromScript {
    aNumber {jsonNumber 0.123}
    aNumberAsString {jsonString 0.123}
    aString {jsonString "this is a string"}
    "a key with spaces" {jsonString "It's possible."}
    aArray {
        # Any Tcl commands are allowed including proc calls
        foreach value {foo bar grill} {
            jsonString $value
        }
    }
}
puts [$resultJSON asJSON -indent 4]
```

This is the output:

```
{
    "aNumber": 0.123,
    "aNumberAsString": "0.123",
    "aString": "this is a string",
    "a key with spaces": "It's possible.",
    "aArray": [
        "foo",
        "bar",
        "grill"
    ]
}
```

Creating JSON – the better way II

- Let me stress: Create your JSON like it should look like at the end
- The argument to `appendFromScript` is an ordinary Tcl script
- Inside that scripts you may use any Tcl command: loops, conditions, proc calls, read of external files, database queries ...
- You can factor out parts of your JSON creation to procs
- The wiki has a more elaborated example and discussion (thanks to rattleCAD an others):
  https://wiki.tcl-lang.org/page/build+JSON+with+tdom

Finis.

Questions?