ChatScript JSON Manual

© Bruce Wilcox, gowilcox@gmail.com brilligunderstanding.com

Revision 3/05/2016 cs6.2c

Real World JSON

JSON (JavaScript Object Notation) is an open standard format using human-readable text to transmit data objects over the web. It is a common standard largely replacing XML which is too wordy and hard to read. JSON has two datatypes that represents collections of values, the array and the object.

A JSON array is a list of JSON entities separated by commas and placed within [], e.g.,

[A 2 [help life] []]

Indices of an array start at 0, so the above has as values:

[0] = A [1] = 2 [2] = an array of 2 values [3] = an empty array

Note that arrays can hold values of different types. The JSON types are array, object, number, string (enclosed in doublequotes), and primitives (text without doublequotes that cannot contain any whitespace). Array values are ordered and always retain that order.

A JSON object is a list of key-value pairs separated by commas and placed within {}, e.g.,

{ “key1”: 1, “bob”: help, “1”: 7, “array”: [1 2 3] “object12”: {} }

Each key must be encased in quotes and joined to an ending colon. Whitespace separates the colon from the value. Again types can be mixed in the values. {} is the empty object. Key-value pairs have no guaranteed order and may shuffle around if you manipulate the structure.

You can nest arrays and objects inside each other.

ChatScript & JSON

JSON is an excellent language to represent more complex ChatScript facts as well as interact with the web. ChatScript can convert back and forth between JSON the text string passed over the web and ChatScript facts that represent the structure internally. If you tried to create facts using CreateFact, you would find making the data shown below extremely difficult and non-obvious. But as JSON, it is easy to create facts to represent the structure and to access pieces of it.

```
{
 "firstName": "John",
 "lastName": "Smith",
 "age": 25,
 "address": {
  "streetAddress": "21 2nd Street",
  "city": "New York",
  "state": "NY",
  "postalCode": "10021-3100"
 },
 "phoneNumbers": [
  {
   "type": "home",
   "number": "212 555-1234"
  },
  {
```

```
    "type": "office",
    "number": "646 555-4567"
   }
  ],
  "children": [],
  "spouse": null
}
```

Note that JSON has no mechanism for sharing JSON subtrees. Hence anytime you create a json fact structure in CS, the facts will all be unique.

JSONFLAGS - Optional 1st arg to some JSON routines
Routines that will create facts for JSON will by default create them as transients (they die at end of volley unless you work to save them). You can override this default by saying *permanent* or *transient.* This applies to ^jsonopen, ^jsonparse, ^jsoncreate, ^jsonobjectinsert, ^jsonarrayinsert, ^jsoncopy.

^jsonparse({JSONFLAGS} string) – string is a JSON text string (as might be returned from a website) and this parses into facts. It returns the name of the root node JSON composite. This name will look like this:

ja-0 – a json array numbered 0.
jo-0 – a json object numbered 0.

As new JSON composites are created during a volley, the numbers increase to keep them all distinct. JSON composites are all created as transient facts and will die at the end of the volley unless you do something to explictly keep them (typically ^jsongather into some factset and then saving that OR using that to remove all their transient flags OR using ^delete() to destroy the facts in the set). If you are keeping JSON across volleys, you should use the optional Json flags argument to make sure numbering never collides (normally the numbers start back at 0 for each new volley).

JSON has stricter requirements on its format than CS does. While CS will output strict JSON, you can input slack JSON. You do not need to put commas between elements of an array or object. And you do not need to put quotes around a key name. So the following is legal text form:

[a b {bob: 1 sue: 2}]

Formally JSON has specific primitives named true, false, and null, that systems may care about, but to CS they are nothing special. Numbers in JSON can be integer or have a decimal point and an exponent. CS does not currently support exponent notation, so you can only do:

1325 566.23

JSON originally required a JSON string be a composite of some kind. Now it allows any JSON type to be a JSON document. CS sticks to the original requirement, because passing around a single value as JSON is pretty useless.

Once you have a JSON fact structure, you can go the opposite direction and convert it back into a string or display it visually.

CS accepts extended JSON syntax for parsing into a json fact structure. Any place you have an object value, you can refer to a ChatScript user or match variable and that value will be substituted in. E.g.,

^parsejson("{ a: $var, b: _0 }")

Note you should use a regular quoted string and not a function string like ^"*{ a: $var, b: _0 }"*. If you use a function string, substitution will happen before calling ^parsejson. Which might be a problem if you had something like this:

^parsejson("{ a: $var, b: _0aba }").

where you ended the value of be to be "_0aba".

Also, you can use json dereference operators to take apart an existing json structure and use values of it in the current one. If $$y points to a json structure, then

^parsejson("{ a: $var, b: _0.e[2] }")

would find a json object reference on _0, get the e field, and get the 3rd array value found there.

^JSONFormat(string) – Because technically JSON requires you put quotes around field names (though various places ignore that requirement) and because CS doesn't, the function takes in a slack json text string and outputs a strict one.

Accessing JSON structures

^jsonpath(string id) - string is a description of how to walk JSON. Id is the name of the node you want to start at (typically returned from ^JSONOPEN or ^JSONPARSE). Array values are accessed using typical array notation like *ja-1[3]* and object fields using dotted notation like *jo-7.id* . A simple path access might look like this: *[1].id* which means take the root object passed as id, e.g., *ja-1*, get the 2nd index value (arrays are 0-based in JSON). That value is expected to be an object, so return the value corresponding to the id field of that object. In more complex situations, the value of id might itself be an object or an array, which you could continue indexing like *[1].id.firstname.*

Of course you don't always have to start at the root. If you know what you want is going become object 7 eventually, you could directly say *.id* given *jo-7* and the system would locate that and get the *id* field. Likewise if you know that key names are somehow unique, you could query for them directly using

^query(direct_v ? verbkey ?)

Or even if the key is not unique you can restrict matches to facts having the JSON_OBJECT_FACT flag.

^query(directflag_v ? verbkey ? 1 ? ? ? JSON_OBJECT_FACT)

Be aware that when ^jsonpath returns the value of an object key, when the value is a simple word, it just returns the word without doublequotes (since CS just stores information as a single word). But if the value contains whitespace, or JSON special characters, that may mess up if you pass it to ^JSONFormat. You can get ^jsonpath to return dangerous data as a string with double quotes around it if you add a 3rd argument "safe" to the call.

^jsonpath(".name" $$jsonobject safe)

^length(jsonid) – returns the number of top-level members are in a json array or object.

Printing JSON structures

^jsonwrite(name) – name is the name from a json fact set (either by ^JSONPART, ^JSONOPEN, or some query into such structures). Result is the corresponding JSON string (as a website might emit), without any linefeeds.

^jsontree(name {depth}) – name is the value returned by ^JSONPARSE or ^JSONOPEN or some

query into such structures. It displays a tree of elements, one per line, where depth is represented as more deeply indented. Objects are marked with { } as they are in JSON. Arrays are marked with []. The internal name of the composite is shown immediately next to its opening punctuation. Optional depth number restricts how deep it displays. 0 (default) means all. 1 is just top level.

JSON structure manipulation

You can build up a JSON structure without using ^JSONparse if you want to build it piece by piece. And you can edit existing structures.

^jsoncreate({JSONFLAGS} type) Type is either array or object and a json composite with no content is created and its name returned. See ^jsonarrayinsert, ^jsonobjectinsert, and ^jsondelete for how to manipulate it. See writeup earlier about optional json flags.

^jsonarrayinsert({JSONFLAGS} arrayname value) Given the name of a json array and a value, it adds the value to the end of the array. See writeup earlier about optional json flags.

^jsonarraysize(name) Given the name of a json array, count how many elements it has.

^jsoncopy(name) Given the name of a json structure, makes a duplicate of it.

^jsonobjectinsert({JSONFLAGS} objectname key value) inserts the key value pair into the object named. The key does not require quoting. Inserting a json string as value requires a quoted string. Duplicate keys are allowed but not advised (standards differ on legality). See writeup earlier about optional json flags.

^jsondelete(factid) Takes the given json fact (eg a member of a json array or object) and deletes that fact and any nested json arrays or objects referred to. If the fact is a json array fact, all later json array values will be renumbered 1 lower.

^jsongather(factset jsonid) takes the facts involved in the json data (as returned by ^jsonparse or ^jsonopen) and stores them in the named factset. This allows you to remove their transient flags or save them in the users permanent data file.

WEB JSON

^jsonopen({JSONFLAGS} kind url postdata header) – this function queries a website and returns a JSON datastructure as facts. It uses the standard "CURL" library, so it's arguments and how to use them are generally defined by CURL documentation and the website you intend to access. See writeup earlier about optional json flags.

Kind is POST, GET, POSTU, GET U, PUT corresponding to the usual meanings of Get and Post and url-encoded forms. **Url** is the url to query. **Postdata** is either "" if this is not a post or is the data to send as post or put. **Header** is any needed extra request headers or "". Multiple header entries must be separated by a tilde. A sample call might be:

```
$$url = "https://api.github.com/users/test/repos"
$$user_agent = ^"User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)"
^jsonopen(GET $$url "" $$user_agent)
```

where GitHub requires user-agent data.

As an example of a complex header value you might create neatly,

```
$header =   ^"Authorization: 8daWs-dwQPpXkuzJO0o
            ~Accept: application/json
            ~Accept-Encoding: identity,*;q=0
            ~Accept-Language: en-US,en;q=0.5
            ~Cache-Control: no-cache
            ~Connection:  close
            ~Host: Chatscript
            ~User_Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:42.0) "
```

ChatScript will make each line have a single space separator between line continuations.
And JsonOpen will correctly get the header elements that do not include that spacing.

The results are a collection of facts representing the JSON parse tree and the value of the function is the root JSON value. The JSON elements that can be represented are arrays, objects, JSON strings, and primitives (numbers, true, false, null). JSON arrays are named “ja-n” where “n” is a unique index. JSON objects are similarly named “jo-n”. Unlike JSON, which makes a distinction between primitives and strings, in ChatScript those things are all strings and are not quoted.
So a JSON string like this:

```
[ {“id”: 1 “value”: “hello”}  {“id”: 2 “value”: “bye”} ]
```

returns this value: *ja-1* and these facts. The facts have flags on them which you can use in queries. You may not have any need to use these flags, so maybe you will just ignore their existence.

```
( ja-1 0 jo-1)   #JSON_ARRAY_FACT #JSON_OBJECT_VALUE
(jo-1 id 1)   #JSON_OBJECT_FACT #JSON_PRIMITIVE_VALUE
(jo-1 value hello )  #JSON_OBJECT_FACT #JSON_STRING_VALUE
( ja-1 1 jo-2  ) #JSON_ARRAY_FACT #JSON_OBJECT_VALUE
(jo-2 id 2  )  #JSON_OBJECT_FACT #JSON_PRIMITIVE_VALUE
(jo-2 value bye)   #JSON_OBJECT_FACT #JSON_STRING_VALUE
```

Using queries, you could get all values of an array. Or all fields of an object. Or all JSON facts where the field is the id. You could manually write script to walk the entire tree. But more likely you will use ^JSONPATH to retrieve specific pieces of data you want. For example, you could do a query on the value returned by ^JSONOPEN as subject, to find out how many array elements there are. Then use ^JSON path to walk each array element to retrieve a specific field buried within.

Note- for things like primitive null, null arrays, null strings, null objects, these are represented as “null” and the corresponding fact flag tells you the kind of value it is.

You can also ask CS to show those out visually using ^JSONtree.

Note that the facts created are all transient and disappear at the end of the volley unless you have forced them to stay. Forcing them to stay is generally a bad idea because it will congest your user topic data file, slowing it down or exceeding its capacity, and because those facts may then collide with new facts created by a new ^JSONOPEN on a new volley. The array and object ids are cleared at each volley, so you will be reusing the same names on new unrelated facts.

Using the flag values, it is entirely possible to reconstruct the original JSON from the facts (if the root is an array or object because otherwise there are no facts involved), but I can't think of use cases at

present where you might want to.

You cannot compile CS on LINUX unless you have installed the CURL library. For Amazon machines that means doing this:

sudo yum -y install libcurl libcurl-devel

On some other machines that doesn't install library stuff and maybe you need

sudo apt-get install libcurl3 libcurl3-gnutls libcurl4-openssl-dev

JSON & Out-of-band output data

Out-of-band data in ChatScript is signaled by the output beginning with data enclosed in [].
Which might be confusing, since JSON uses [] to denote an array. Standalone ChatScript contains a built-in handler for OOB data and if you pass it JSON at the start of output, it will swallow it and not display it (unless you turn on OOB display). Similarly, std webpage interfaces connecting to ChatScript do likewise. So if you want to see this information, you should put something in the output at the start which is NOT the JSON data. Anything will do. The only time you might actually need the JSON clean at the beginning is from some special purpose application, and in that case you will write your own OOB handler anyway (or not have one).

JSON & Out-of-band input data

OOB data into ChatScript is similarly signaled by being at the start of input, with data enclosed in [], followed typically by the user's actual input.
The ChatScript engine reacts specially to OOB incoming data in that it will be careful to not treat it like ordinary user chat. Tokenization is done uniquely, spell-checking, pos-tagging, parsing, named entity merging etc are all turned off and the data becomes its own sentence (the user's actual input generates more sentences to CS as input). OOB data is then processed by your script in any way you want. So one clever thing you can do is pass in JSON data within the OOB to get temporary facts into your app during a volley. Input might look like this:

[[a b { “bob”: 1, “suzy”: 2 }]] What is your name?

You can pattern match the oob section of the input as follows:

u: (\[_*) $$tmp = ^jsonparse('_0)

_0 will contain an excess right bracket (the end of the oob message), but that won't bother ^jsonparse.

Representing JSON in CS facts is more than just a bunch of subject-verb-object facts linked together. The facts have typing bits on them that describe the structure and arrays have index values that must remain consistent. Therefore you should not create and alter JSON fact structures using ordinary CS fact routines like ^createfact and ^delete. Instead use the JSON routines provided.