Manuel JSON et ChatScript

© Bruce Wilcox, mailto:gowilcox@gmail.com www.brilligunderstanding.com
Revision 9/24/2017 cs7.55

traduction Mathieu Rigard 02/12/2017 pour Utopia : m.rigard@utopia-french.tech

Ce qu’est le JSON

JSON (JavaScript Object Notation) est un format standard ouvert utilisant un texte lisible par les humains pour transmettre des données sur le Web sous forme d’objets. C'est une norme qui remplace en grande partie le XML qui est trop verbeux et difficile à lire. JSON possède deux types de données qui représentent des ensembles de valeurs, le tableau et l'objet. Un tableau JSON est une liste d'entités JSON séparées par des virgules et placées entre [], par exemple:

```
[ A 2 [ aide  vie] [] ]
```

Les indices d'un tableau commencent à 0, donc l’exemple ci-dessus a comme valeurs: [0] = A [1] = 2 [2] = un tableau de 2 valeurs [3] = un tableau vide.

Notez que les tableaux peuvent contenir des valeurs de différents types (puisqu’en fait tout en interne est de type texte). Les types JSON sont le tableau(array), l'objet(object), le nombre(number), la chaîne(string) (insérées entre guillemets) et les primitives (texte sans guillemets qui ne peuvent contenir aucun espace). Les valeurs de tableau sont ordonnées et conservent toujours cet ordre. Un objet JSON est une liste de paires de clés séparées par des virgules et placées entre {}, par exemple :

```
{
    "key1": 1,
    "bob": "help",
    "1": 7,
    "array": [1 2 3],
    "object12": {}
```

```
}
```

Chaque clé doit être mise entre guillemets et suivie d’une marque de fin de colonne. L'espace blanc sépare la colonne de la valeur qui lui est jointe. Différents types peuvent venir se placer en valeurs. {} est l'objet vide. Les paires de clés n'ont pas d'ordre garanti et peuvent se déplacer si vous manipulez la structure. Vous pouvez insérer des tableaux et des objets les uns dans les autres.

ChatScript & JSON

JSON est un excellent langage pour représenter les Faits complexes de ChatScript et interagir avec le Web. ChatScript peut convertir dans un sens et dans l’autre la chaîne de texte JSON passée par le Web et les Faits ChatScript qui représentent la structure en interne. Si vous avez essayé de créer des Faits en utilisant ^CreateFact, vous trouverez que les données ci-dessous sont extrêmement difficiles et non intuitives. Mais vec JSON, il est facile de créer des Faits pour représenter la structure et d’accéder à des parties de celle-ci.

```
{
    "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
}
```

Notez que JSON n'a aucun mécanisme pour le partage de sous-arborescence JSON. Par conséquent, chaque fois que vous créez une structure de Faits JSON dans CS, ces Faits seront tous uniques.

Accéder au Web avec JSON

Vous apprendrez comment créer des structures JSON plus bas. L'autre outils pour rendre CS compatible avec le Web est ^jsonopen. Pour voir à quel point il est facile de dialoguer avec le Web, regardez ce code.

```
# La manière la plus propre pour créer des objets JSON
$_var = ^jsoncreate(transient object)
$_var.fieldname = $myvar

Une autre méthode mais limitée à une sortie tampon max (~80K):
$_var = ^jsonparse(^"{ fieldname: $myvar }")

Et maintenant pour envoyer les donnée :
$_url = ^"http://myHost:myIP"
$_userAgent = ^"User-Agent: %bot, ChatScript”
$_header = ^" ~Accept: application/json
                $_userAgent
                ~Content-Type: application/json "
$_response = ^jsonopen(transient POST $_url $_data $_header $_userAgent)
```

JSONFLAGS – Option : 1er argument pour des routines JSON

Les routines qui créeront des Faits pour JSON les créent par défaut transitoires (ils meurent à la fin de la volée, sauf si vous œuvrez à les sauvegarder). Vous pouvez remplacer ce défaut en précisant permanent ou transient. Ça s'appliqueR à
^jsonopen, ^jsonparse, ^jsoncreate, ^jsonobjectinsert, ^jsonarrayinsert, ^jsoncopy.
Vous pouvez aussi ajouter le drapeau(flag ou argument de balises) safe to ^jsonparse, ^jsonobjectinsert, ^jsonarraydelete. You can also add le flag unique à ^jsonarrayinsert. Vous pouvez aussi ajouter duplicate à ^jsonobjectinsert.
Lorsque plusieurs drapeaux sont souhaités, mettez-les dans une chaîne simple, "DUPLICATE PERMANENT". Non sensible à la casse.
Lorsque vous souhaitez ajouter une référence à une faction normale (comme celle retournée par ^createfact), vous pouvez ajouter le drapeau AUTODELETE
Vous pouvez également attribuer utilisateurs de Flag par liste USER_FLAG1, par l'intermédiaire de USER_FLAG4 étant un drapeau lui-même. Le Fait JSON aura ce Flag que vous pouvez utiliser conjointement avec ^query pour limiter les correspondances qui peuvent être trouvées.

^jsonparse({JSONFLAGS} string)

string est une chaîne de texte JSON (comme cela pourrait être retourné à partir d'un site web) et cela donne un Faits. Il renvoie le nom du nœud racine JSON composite. Ce nom ressemblera à ceci:

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

Notez que les noms ci-dessus sont vrais uniquement pour les objets PERMANENT JSON. Ceux qui sont transitoires seront nommées ja-t0 and jo-t0.

Comme les nouveaux JSON composés sont créés pendant une volée, les nombres augmentent pour les garder distincts. Les JSON composites sont tous créés comme des Faits transitoires et mourront à la fin de la volée, à moins que vous ne fassiez quelque chose pour les garder explicitement (habituellement, dans certains cas, puis en les sauvegardant ou en utilisant ceux-ci pour supprimer tous leurs drapeaux transitoires OU en utilisant ^delete() pour détruire les faits dans l'ensemble).

Si vous maintenez les JSON de volée en volée, vous devriez utiliser le Flag Json optionnel pour vous assurer que la numérotation ne se téléscope jamais (normalement, les chiffres commencent à 0 pour chaque nouvelle volée).

JSON a des exigences plus strictes sur son format que CS.

Comme CS produira un JSON stricte, vous pouvez entrer JSON simplifié. Vous n'avez pas besoin de mettre des virgules entre les éléments d'un tableau ou d'un objet. Et vous n'avez pas besoin de mettre des guillemets autour d'un nom clé. Donc, ce qui suit est une formulation acceptable :
`[a b {bob: 1 sue: 2}]`

Formellement, JSON a des primitives spécifiques nommées true(vraies), false(fausses) et null(nulles), que le systèmes prend en compte, mais pour CS, ils ne veulent rien dire en particulier. Les nombres dans JSON peuvent être entiers un entier à virgule ou un exposant. CS ne supporte actuellement aucune notation d'exposant, donc vous ne pouvez faire que: 1325 ou 566.23.

JSON exigeait à l'origine qu'un string JSON soit un peu composite. Maintenant, il permet à tout type JSON d'être un document JSON. CS adhère à l'exigence initiale, car passer une seule valeur comme JSON est assez inutile.

Une fois que vous avez une structure de Fait JSON, vous pouvez aller dans le sens inverse et la convertir en une chaîne ou l'afficher visuellement.

CS accepte la syntaxe JSON étendue pour « parser »(convertir/analyser) vers la structure d’un Fait json. Où que vous ayez une valeur d'objet, vous pouvez vous référer à un utilisateur ChatScript ou à une variable correspondant et cette valeur sera remplacée. Ex :

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

Notez que vous devez utiliser une chaîne régulière entre guillemets et non une chaîne de fonction : ^"{ a: $var, b: _0 }". Si vous utilisez une chaîne de fonction, la substitution se produira avant d'appeler ^jsonparse. Ce pourrait être un problème si vous aviez quelque chose comme ceci:
`^jsonparse("{ a: $var, b: _0aba }").`

where you wanted the value of b to be "_0aba". Had you used an active string, the _0 would have been replaced with its contents.
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

```
^jsonparse("{ a: $var, b: $_y.e[2] }")
```

would find a json object reference on $_y, get the e field, and get the 3rd array value found there. An initial argument of safe will locate the correct end of the data for json parsing, allowing you to pass excess data. This is important for passing in json data in OOB information. OOB information comes in [] and Json itself uses [] at times, so it would be impossible to find the correct end of pattern with normal pattern matching. Instead just do:

```
u: ( \[ _* ) and call ^jsonparse(safe _0)
```

This will figure out where the OOB marker actually is and only parse within it. You can add NOFAIL before the string argument, to tell it to return null but not fail if a dereference path fails cannot be found.

```
^jsonparse(transient NOFAIL "{ a: $var, b: _0.e[2] }")
```

Note: You cannot send JSON text as normal input from a user because CS tokenization is set to handle human input. So doing something like this as input will not work:

```
doLogin {"token": "myid", "accounts": ["whatever"]}
```

Instead you should call a website using ^jsonopen which will automatically convert returned JSON data into internal CS data. Or you can pass JSON data on input via OOB notation:

```
[ {"token": "myid", "accounts": ["whatever"]}]  User message
```

and then have a pattern to grab the OOB data and call jsonparse with it (using SAFE as a parameter). OOB input is not subject to human tokenization behavior, spellchecking, etc.
Note: There is a limit to how much JSON you can pass as OOB data nominally, because it is considered a single token. You can bypass this limit by asking the tokenizer to directly process OOB data, returning the JSON structure name instead of all the content. Just enable #JSON_DIRECT_FROM_OOB on

the $cs_token value and if it finds OOB data that is entirely JSON, it will parse it and return something like jo-t1 or ja-t1 in its place. Eg. [{ "key": "value}] will return tokenized as [jo-t1].
Note: ^jsonparse autoconverts backslash-unnnn into corresponding the utf8 characters.

jsonformat(string)

Parce que techniquement, JSON exige que vous mettiez des guillemets autour des noms de champs (bien que divers endroits ignorent cette exigence) et parce que CS ne le fait pas pour vous, la fonction prend une chaîne de texte json et envoie une version stricte.

Accéder à la structure du JSON

^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 [3] and object fields using dotted notation. 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. You can walk an array by using [$_index] and varying the value of $_index. When you access an array element, you have to quote the text because it consists of multiple tokens to CS which breaks off [and]. If you are just accesing an object field of something, you can quote the string or just type it direct

```
^jsonpath(.id $object2)
^jsonpath(".id" $object2)
```

Of course you don't always have to start at the root. If you know what you want is going to 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)
```

^jsonpath

Can also return the actual factid of the match, instead of the object of the fact. This would allow you to see the index of a found array element, or the json object/array name involved. Or you could use ^revisefact to change the specific value of that fact (not creating a new fact). Just add * after your final path, eg

```
^jsonpath(.name* $$obj)
^jsonpath(.name[4]* $$obj)
```

Correspondingly, if you are trying to dump all keys and values of a JSON object, you could do a query like this:

```
@0 = ^query(direct_s $_jsonobject ? ?)
^loop()
{
    _0 = ^first(@0all)
    and then you have _1 and _2 as key and value
}
```

If you need to handle the full range of legal keys in json, you can use text string notation like this ^jsonpath(."st. helen".data $tmp).
You may omit the leading . of a path and CS will by default assume it

```
^jsonpath("st. helen".data $tmp)
```

L’accès direct via les variables JSON $myvar.field et `$myvar[]

Si une variable contient une valeur d'objet JSON, vous pouvez définir directement et obtenir des champs de cet objet en utilisant la notation avec un point. Ceci peut être un nom de champ statique fixe que vous donnez ou une valeur variable: $myvar.$myfield est légal.
La notation avec point est plus propre et plus rapide que ^jsonpath et jsonobjectinsert et pour GET, a l'avantage de ne jamais échouer, elle ne renvoie NULL que si elle ne peut pas trouver le champ. Si le chemin ne contient pas d'objet json au niveau inférieur au sommet, créera automatiquement une affectation et aura la même propriété transitoire/permanente que l'objet directement contenu. Si la variable de niveau supérieur n'est pas actuellement un objet, l'affectation échouera. CS ne créera pas d'objet pour vous parce qu'il ne sait pas s'il doit être transitoire ou permanent.

```
$x = $$obj.name.value.data.side
$$obj.name.value.data.side = 7
```

De même, vous pouvez accéder aux tableaux JSON en utilisant la notation de tableau:

```
$x = $$array[5]
$x = $$array[$_tmp]
$$obj.name[4] += 3
$x.foo[] = Bruce
```

Si foo est actuellement indéfini, le système créera un tableau JSON pour vous, avec une permanence qui correspond à l'objet JSON de $x. Vous ne pouvez pas faire $x[] et que cela fonctionne car, au niveau supérieur, le système ne connaît pas la permanence à utiliser. Une fois qu'il y a un tableau JSON dans $x.foo, les affectations avec foo[] ajouteront des éléments au tableau. Vous ne pouvez pas désigner l'index, ce sera le prochain index successif.
La seule restriction sur les tableaux est que vous ne pouvez pas ajouter une nouvelle valeur d'index de tableau sans utiliser ^jsonarrayinsert car vous n'êtes pas autorisé à créer des indices inconnus.

NOTE : JSON est normalement une structure non récursive sans pointeurs partagés. Mais ChatScript vous permet de stocker les références aux structures JSON dans plusieurs endroits d'autres structures JSON. Cela a ses inconvenants. Il ne présente aucun problème lors de la transcription au texte d'un site Web utilisant ^jsonwrite. Et quand vous avez quelque chose comme ceci:

```
$x = ^jsoncreate(object)
$y = ^jsoncreate(object)
$x.field = $y
$x.field1 = $y
$x.field = null
```

En supposant qu'une structure JSON n'est pas disponible dans plusieurs endroits, l'attribution de NULL (ou de toute autre valeur) à un champ qui possède déjà une structure JSON entraînera normalement l'élimination complète de l'ancienne structure de valeur, car sa seule référence est supprimée. Et le système vérifie et supprime la structure s'il n'est pas mentionné par un autre champ JSON. Mais il y a des limites. Le système n'a aucune idée du fait que vous ayez un pointeur dans une variable. Ou si cela fait partie d'une séquence fausses routes pathologique comme celle-ci:

```
$x = ^jsoncreate(object)
$y = ^jsoncreate(object)
$x.field = $y
$y.field = $x
$x.field = null
```

Les deux structures se désignent, chacune une fois. Ainsi, assigner NULL va tuer les deux structures.

L'attribution NULL supprime entièrement une clé JSON. L'assignation "" et ^"" définira le champ sur le JSON littéral.

^length(jsonid)

Renvoie le nombre de membres de haut niveau dans un tableau json ou un objet.

Afficher une structure JSON

^jsonwrite(name)
name is the name from a json fact set (either by ^jsonpart, ^jsonopen, orsome 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.

Manipulation d'une structure JSON

Vous pouvez créer une structure JSON sans utiliser ^jsonparse si vous souhaitez la construire pièce par pièce. Et vous pouvez modifier les structures existantes.

^jsoncreate({JSONFLAGS} type)
Le type est un tableau ou un objet, un json composite sans contenu est crééde et son nom est renvoyé. Voir ^jsonarrayinsert, ^jsonobjectinsert, et ^jsondelete pour savoir comment le manipuler. Voir plus haut le sujet sur les drapeaux json optionnels.

^jsonarrayinsert({JSONFLAGS} arrayname value)
Compte tenu du nom d'un tableau json et d'une valeur, il ajoute la valeur à la fin du tableau. Voir plus haut le sujet sur les drapeaux json optionnels. Si vous utilisez le drapeau unique, si la valeur existe déjà dans le tableau, aucun duplicata ne sera ajouté.

^jsonarraydelete([INDEX, VALUE] arrayname value)
This deletes a single entry from a JSON array. It does not damage the thing deleted, just its member in the array.

- If the first argument is INDEX, then value is a number which is the array index (0 ... n-1).
- If the first argument is VALUE, then value is the value to find and remove as the object of the json fact.

You can delete every matching VALUE entry by adding the optional argument ALL. Like: ^jsonarraydelete("INDEX ALL" $array 4)
If there are numbered elements after this one, then those elements immediately renumber downwards so that the array indexing range is contiguous.

If the key has an existing value then if the value is a json object it will be recursively deleted provided its data is not referenced by some other fact (not by any variables). You can suppress this with the SAFE flag. ^jsonarraydelete(SAFE $obj $key).

^jsonarraysize(name)
deprecated in favor of ^length

^jsoncopy(name)
Compte tenu du nom d'une structure json, en fait un double. Si ce n'est pas le nom d'une structure json, il renvoie simplement ce que vous passez.

^jsonobjectinsert({JSONFLAGS} objectname key value)
insère les valeurs de la paire de clé dans l'objet nommé. La clé ne nécessite pas de guillemets. L'insertion d'une chaîne json comme valeur nécessite des guillemets. Les clés en double sont ignorées à moins que le 1er argument optionnel DUPLICATE ne soit donné. Voir plus haut le sujet sur les drapeaux json optionnels.
Si la clé a une valeur existante et que DUPLICATE n'est pas un facteur, alors si la valeur est un objet json, elle sera supprimée de façon récursive à condition que ses données ne soient pas référencées par un autre fait (et non par des variables). Vous pouvez supprimer ceci avec le drapeau SAFE. Jsonobjectinsert (SAFE $ obj $ key null).

^jsondelete(factid)
Obsolète, remplacé par ^delete

^jsongather({fact-set} jsonid {level})
prend les faits impliqués dans les données de json (comme retourné par ^jsonparse ou ^jsonopen) et les stocke dans le compte-rendu nommé.

Cela vous permet de supprimer leurs drapeaux transitoires ou de les enregistrer dans le fichier de données permanent des utilisateurs.
Vous pouvez omettre les Faits comme argument si vous utilisez une instruction d'affectation: @ 1 = ^ jsongather (jsonid)
^Jsongather regroupe normalement tous les niveaux de données de manière récursive. Vous pouvez limiter la distance à parcourir en fournissant le niveau. Le niveau 0 est tout. Le niveau 1 est le niveau supérieur de données. Etc.
.

^jsonlabel(label)
attribue une séquence de texte à ajouter jo- et ja- créés par la suite. Voir le manuel des fonctions du système.

^jsonreadcsv(TAB filepath {'^fn})
lit un fichier tsv (fichier tableur délimité par des tabulations) et renvoie un tableau JSON qui le représente. Les lignes sont tous des objets dans un tableau. La ligne est un objet où les champs non vides sont donnés sous forme d'index de champ. Le premier champ est 0. Les champs vides sont ignorés et leur nombre est supprimé.

Si un troisième paramètre facultatif d'un nom de fonction est donné, le code ne crée pas de structure JSON à renvoyer. Au lieu de cela, il appelle la fonction, chaque champ d'une ligne étant un argument. Ceci est analogue à: le mode document en ce sens que vous pouvez lire de grandes quantités de données en une seule volée et peut-être besoin d'utiliser ^memorymark et ^memoryfree pour gérer le problème.

^jsonundecodestring(string)
supprime tous les marqueurs d'échappement json à la normale pour une sortie utilisateur possible. Cela traduit \n à la nouvelle ligne, \r à un retour chariot, \t à une tabulation et \" à une simple apostrophe.

WEB JSON

^jsonopen({JSONFLAGS} kind url postdata header {timeout})
cette fonction interroge un site Web et renvoie des données structurées JSON comme des Faits. Elle utilise la bibliothèque standard CURL, donc ses arguments et sa façon de les utiliser sont généralement définis par la documentation CURL et le site Web auquel vous souhaitez accéder. Voir le manuel des fonctions du système.

paramètre	description
kind	is POST, GET, POSTU, GETU, PUT, DELETE correspondant aux significations habituelles de Get et Post et des formulaires url-encoded.
url	est l'url à interroger
postdata	"" s'il ne s'agit pas d'un POST, ou si c’est la donnée à envoyer en POST ou PUT
header	est-ce qu'il y a des en-têtes de requêtes supplémentaires ou ""? Plusieurs entrées d'en-tête doivent être séparées par un tilde ~
timeout	limitation de secondes optionnelle pour la connexion, puis pour le transfert. Autre règle $cs_jsontimeout

Remarque: 'postdata' peut être un nom de structure JSON simple, auquel cas le système exécutera automatiquement un ^jsonwrite sur celui-ci et envoie les données textuelles comme données. Actuellement limité à 500K de taille de la mémoire tampon interne.

Un exemple d'appel pourrait être :

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

^jsonopen(GET $$url "" $$user_agent)
```

où GitHub requiert des données utilisateur-agent. Comme exemple d'une valeur d'en-tête complexe, vous pouvez créer de manière soignée,

```
$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.

fact	associated flags
(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

fact	associated flags
	#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 ^jsonpath 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 via permanent. 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
```

System variables %httpresponse will hold the most recent http return code from calling ^jsonopen.
If you call ^jsonopen(direct ... then the result will not be facts, but the text will be directly shipped back as the answer. Be wary of doing this if the result will be large (>30K?) since you will overflow your buffer without being checked.
^JSONopen automatically url-encodes headers and urls

JSONOpen and proxy servers

If you need JSONOpen to run thru a proxy server, these are the CS variables you need to set up: $cs_proxycredentialsshould be your login data,
e.g. myname:thesecret. $cs_proxyserver' is the server address,
e.g.,http://local.example.com:1080`. '$cs_proxymethod' are bits listing the authorization method to use. They come from the LIBCURL so you should OR together the bits you want. Bit 1 is the most basic choice of name and password.
Read-https://curl.haxx.se/libcurl/c/CURLOPT_HTTPAUTH.html

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.

Practical Examples

Objects

The write jsonwrite and json tree print out different views of the same data..

```
u: (-testcase1) $_jsonObject = ^jsoncreate(object)
    ^jsonobjectinsert( $_jsonObject name “some name” )
    ^jsonobjectinsert( $_jsonObject phone “some number” )
    ^jsonwrite ( $_jsonObject ) \n
    ^jsontree ( $_jsonObject )\n
```

Note in this next example how to escpe a json string with ^''. This makes creating json objects from static data very intuitive and clear.

```
u: (-testcase2) $_tmp = ^jsonparse( ^'{name: "Todd Kuebler", phone: "555.1212"}' )
    ^jsonwrite( $_tmp ) \n
    ^jsontree( $_tmp ) \n
    name: $_tmp.name, phone: $_tmp.phone
```

This example shows the . notation access of data inside an json object in chatscript. This is probably the most intuitive way of interacting with the data.

```
u: (-testcase3) $_tmp = ^jsoncreate(object)
    $_tmp.name = "Todd Kuebler"
    $_tmp.phone = "555-1212"
    ^jsonwrite( $_tmp ) \n
    ^jsontree( $_tmp ) \n
    name: $_tmp.name, phone: $_tmp.phone
```

Arrays of objects

In the example below, we add two items into an array of objects and we display the formatted array:

```
u: ( testcase4 )
    # create a phoneBook as an array of structured items (objects)
    $_phoneBook = ^jsoncreate(array)

    #
    # add first object in the array
    #
    $_item = ^jsoncreate(object)

    # assign values
    $_item.name = "Todd Kuebler"
    $_item.phone = "555-1212"

    ^jsonarrayinsert($_phoneBook $_item)

    #
    # add a second object in the array
    #
    $_item = ^jsoncreate(object)

    # assign values
```

```
$_item.name = "Giorgio Robino"
$_item.phone = "111-123456789"

^jsonarrayinsert($_phoneBook $_item)

# display JSON tree
^jsontree( $_phoneBook ) \n

#
# print formatted items in the phone book
#
phone book:\n
$_i = 0
$_size = ^length($_phoneBook)
loop($_size)
{
   # print out formatted item
   name: $_phoneBook[$_i].name, phone: $_phoneBook[$_i].phone\n
   $_i += 1
}
```