Package 'RJSONIO'

November 14, 2018

Version 1.3-1.1

Title Serialize R Objects to JSON, JavaScript Object Notation

Description This is a package that allows conversion to and from
data in Javascript object notation (JSON) format.
This allows R objects to be inserted into Javascript/ECMAScript/ActionScript code
and allows R programmers to read and convert JSON content to R objects.
This is an alternative to rjson package. Originally, that was too slow for converting large R objects to JSON
and was not extensible. rjson's performance is now similar to this package, and perhaps slightly faster in some cases.
This package uses methods and is readily extensible by defining methods for different classes, vectorized operations, and C code and callbacks to R functions for deserializing JSON objects to R.
The two packages intentionally share the same basic interface. This package (RJSONIO) has many additional
options to allow customizing the generation and processing of JSON content.
This package uses libjson rather than implementing yet another JSON parser. The aim is to support other general projects by building on their work, providing feedback and benefit from their ongoing development.

Note See http://www.json.org/JSON_checker/test.zip for tests. We now
use code from the libjson project
(http://libjson.sourceforge.net).

License BSD_3_clause + file LICENSE

Imports methods

Collate readJSON.R asVars.R isValid.R json.R simpleHandler.R stream.R

Biarch true

NeedsCompilation yes

Author Duncan Temple Lang [aut, cre] (<https://orcid.org/0000-0003-0159-1546>),
Jonathan Wallace [aut] (aka ninja9578, author of included libjson
sources)

Maintainer ORPHANED

X-CRAN-Original-Maintainer Duncan Temple Lang <duncan@r-project.org>

X-CRAN-Comment Orphaned and corrected on 2018-11-12, after 4 years of no corrections.

Repository CRAN

Date/Publication 2018-11-14 12:47:36 UTC

R topics documented:

`asJSVars` *Serialize R objects as Javsscript/ActionScript variables*

Description

This function takes R objects and serializes them as Javascript/ActionScript values. It uses the specified names in the R call as Javascript variable names. One can also specify qualifiers ('public', 'protected', 'private') and also types. These are optional, but useful, in ActionScript.

Usage

```
asJSVars(..., .vars = list(...), qualifier = character(), types = character())
```

Arguments

`...`	`name = value` pairs where the `value` is an R object that is converted to JSON format and `name` is the name of the corresponding Javascript variable.
`.vars`	this is an alternative to `...` as a way to specify a collection of `name = value` pairs that is already in a list.
`qualifier`	a character vector (recycled as necessary) which is used as qualifiers for the individual ActionScript variables. The values should be public, protected or private.
`types`	either a logical value or a character vector (which is recycled if necessary). If this is `TRUE`, then we compute the Javascript type for each of the R objects (using the non-exported function `jsType`)

Value

A character vector of length 1 giving the variable declarations and initializations.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

toJSON

Examples

```
cat(asJSVars( a =  1:10, myMatrix = matrix(1:15, 3, 5)))
cat(asJSVars( a =  1:10, myMatrix = matrix(1:15, 3, 5), types = TRUE))
cat(asJSVars( a =  1:10, myMatrix = matrix(1:15, 3, 5),
       qualifier = "protected", types = TRUE))
```

---

basicJSONHandler *Create handler for processing JSON elements from a parser*

Description

This function creates a handler object that is used to consume tokens/elements from a JSON parser and combine them into R objects.

This is slow relative to using C code because this is done in R and also we don't know the length of each object until we have consumed all its elements.

Usage

```
basicJSONHandler(default.size = 100, simplify = FALSE)
```

Arguments

default.size	the best guess as to the sizes of the different elements. This is used for preallocating space for elements
simplify	a logical value indicating whether to simplify arrays from lists to vectors if the elements are of compatible types.

Value

update	a function called with a JSON element and used to process that element and add it to the relevant R object
value	a function to retrieve the result after processing the JSON

Author(s)

Duncan Temple Lang

See Also

fromJSON and the handler argument.

Examples

```
h = basicJSONHandler()
x = fromJSON("[1, 2, 3]", h)
x
h$value()
```

Bob *Symbolic constants identifying the type of a JSON value.*

Description

These constants are used by handler functions that are called when a JSON value is encountered by the JSON parser. These identify the type of the JSON value. The values will already have been converted, but the start and end array and object events won't have a type.

Format

A collection of integer values.

Source

`JSON_parser.h` code from http://www.json.org.

References

http://www.json.org.

`fromJSON` *Convert JSON content to R objects*

Description

This function and its methods read content in JSON format and de-serializes it into R objects. JSON content is made up of logicals, integers, real numbers, strings, arrays of these and associative arrays/hash tables using `key : value` pairs. These map very naturally to R data types (logical, integer, numeric, character, and named lists).

Usage

```
fromJSON(content, handler = NULL,
          default.size = 100, depth = 150L, allowComments = TRUE,
           asText = isContent(content), data = NULL,
            maxChar = c(0L, nchar(content)), simplify = Strict,
             nullValue = NULL, simplifyWithNames = TRUE,
              encoding = NA_character_, stringFun = NULL, ...)
```

Arguments

content	the JSON content. This can be the name of a file or the content itself as a character string. We will add support for connections in the near future.
handler	an R object that is responsible for processing each individual token/element within the JSON content. By default, this is NULL and we use the fast libjson parsing approach. Unless you want to customize the processing of the nodes in the tree, use NULL. This can be an R function, a list of functions with class "JSONParserHandler" having update and value elements, or the address of a native (C) routine. In the case of the latter, the data parameter can be used to specify an object that is passed to the C routine each time it is called. This will commonly be an externalptr object.
default.size	a number giving the default buffer size to use for arrays and objects in an effort to avoid reallocating each time we add a new element.
depth	the maximum number of nested JSON levels, i.e. arrays and objects within arrays and objects.
allowComments	a logical value indicating whether to allow C-style comments within the JSON content or to raise an error if they are encountered.
asText	a logical value indicating whether the value of the content argument should be treated as the JSON content, i.e. read directly rather than considered the name of a file.
data	a value that is only used when the value of handler is a native (C) routine. In this case, the value is passed in each call to that C routine by the JSON tokenizer.
maxChar	an integer vector of length 2 giving the start and end offsets in the character string to be processed. This allows the caller to specify a subset of the string to process without explicitly having to make a copy of the substring.
simplify	either a logical value or a number, e.g. the value of the variable Strict (the default). This controls whether we attempt to collapse collections/arrays of homogeneous scalar elements to R vectors. If this is FALSE, no effort to combine scalars is made and they remain as separate list elements. If this is TRUE, then logicals, numbers and strings are collapsed to their common types in the same manner as c. The value Strict does attempt to collapse collections of scalars but only if they are all of the same type, i.e. all strings, all numbers or all logicals. If we want to collapse numbers, but not logicals or characters, we can use StrictNumeric. Similarly, to collapse logicals but not numeric or character collections, we use StrictLogical. And, to collapse only character collections, we use StrictCharacter. If we want to collapse two types but not a third, we add the two values, e.g. StrictLogical + StrictNumeric, or pass them as a vector c(StrictLogical, StrictNumeric). Strict is merely the combination of all 3 of the individual strict variables. Currently this is only implemented when the caller does not provide a handler and in the C code.
nullValue	an R value that is used when we encounter a JSON null value in the JSON content. This can be used to map null to something more R-like such as NA. This can be an arbitrary R object.

`simplifyWithNames`
: a logical value that controls whether we attempt to collapse collections if the elements have names in the JSON content, i.e. a dictionary/associative array. If this is `TRUE`, then we consider collapsing according to the value of `simplify`. If this is `FALSE`, if the collection has names, we do not attempt to simplify.

`encoding`
: the encoding for the content. This is used to ensure the encoding of any resulting strings/character vectors have this encoding. The default for this value is to use the same encoding as the input content.

`...`
: additional parameters for methods.

`stringFun`
: an R function or a compiled routine (by address or name). The purpose of this is to process every string as it is encountered in the JSON content and to either convert return it as-is, or to convert it to a suitable R value. This, for example, might convert strings of the form "/new Date(2313213)/" or "/Date(12312312)/". The result is placed in the R object being generated from the JSON content where the original string would appear. So this allows us to handle strings with a special meaning.

  If this is an R function, it is passed a single argument - the value of the string - and it can return that or any other R object, presumably derived from that original string. If a compiled routine is specified, it can be one of two types. Both take a simple C string. The default type returns a `SEXP`, i.e. an R object. If the class of `stringFun` is either `AsIs` or `NativeStringRoutine`, then that routine must return a C string, i.e. a char *. This will then be converted to an R character vector of length 1, using the default encoding given by `encoding`.

Value

An R object created by mapping the JSON content to its R equivalent.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu

References

http://www.json.org

See Also

`toJSON` the non-exported collector function `{RJSONIO:::basicJSONHandler}`.

Examples

```
fromJSON(I(toJSON(1:10)))

fromJSON(I(toJSON(1:10 + .5)))

fromJSON(I(toJSON(c(TRUE, FALSE, FALSE, TRUE))))
```

```
  x = fromJSON('{"ok":true,"id":"x123","rev":"1-1794908527"}')


   # Reading from a connection. It is a text connection so we could
   # just read the text directly, but this could be a dynamic connection.
  m = matrix(1:27, 9, 3)
  txt = toJSON(m)
  con = textConnection(txt)
  identical(m, fromJSON(con)) # not true! fromJSON() returns just a list.

    # Use a connection and move the cursor ahead to skip over some lines.
  f = system.file("sampleData", "obj1.json", package = "RJSONIO")
  con = file(f, "r")
  readLines(con, 1)
  fromJSON(con)
  close(con)


  f = system.file("sampleData", "embedded.json", package = "RJSONIO")
  con = file(f, "r")
  readLines(con, 1)  # eat the first line
  fromJSON(con, maxNumLines = 4)
  close(con)

## Not run:
if(require(rjson)) {
    # We see an approximately a factor of 3.9 speed up when we use
    # this approach that mixes C-level tokenization and an R callback
    # function to gather the results into objects.

  f = system.file("sampleData", "usaPolygons.as", package = "RJSONIO")
  t1 = system.time(a <- RJSONIO:::fromJSON(f))
  t2 = system.time(b <- fromJSON(paste(readLines(f), collapse = "\n")))
}

## End(Not run)
    # Use a C routine
  fromJSON(I("[1, 2, 3, 4]"),
           getNativeSymbolInfo("R_json_testNativeCallback", "RJSONIO"))

    # Use a C routine that populates an R integer vector with the
    # elements read from the JSON array. Note that we must ensure
    # that the array is big enough.
  fromJSON(I("[1, 2, 3, 4]"),
           getNativeSymbolInfo("R_json_IntegerArrayCallback", PACKAGE = "RJSONIO"),
           data = rep(1L, 5))

  x = fromJSON(I("[1.1, 2.2, 3.3, 4.4]"),
               getNativeSymbolInfo("R_json_RealArrayCallback", PACKAGE = "RJSONIO"),
                data = rep(1, 5))
  length(x) = 4
```

```
  # This illustrates a "specialized" handler which knows what it is
  #  expecting and pre-allocates the answer
  # This then populates the answer with the values.
  # The speed improvement is 1.8 versus "infinity"!

x = rnorm(1000000)
str = toJSON(x, digits = 6)

fromJSON(I(str),
         getNativeSymbolInfo("R_json_RealArrayCallback", PACKAGE = "RJSONIO"),
         data = numeric(length(x)))


  # This is another example of very fast reading of specific JSON.
x = matrix(rnorm(1000000), 1000, 1000)
str = toJSON(x, digits = 6)

v = fromJSON(I(str),
         getNativeSymbolInfo("R_json_RealArrayCallback", PACKAGE = "RJSONIO"),
         data = matrix(0, 1000, 1000))


  # nulls and NAs
fromJSON("{ 'abc': 1, 'def': 23, 'xyz': null, 'ooo': 4}", nullValue = NA)
fromJSON("{ 'abc': 1, 'def': 23, 'xyz': null, 'ooo': 4}", nullValue =  NULL) # default

fromJSON("[1, 2, 3, null, 4]", nullValue = NA)
fromJSON("[1, 2, 3, null, 4]", nullValue = NULL)


 # we can supply a complex object for null if we ever should need to.
fromJSON('[ 1, 2, null]', nullValue = list(a = 1, b = 1:10))[[3]]


# Using StrictNumeric, etc.
x = list(sub1 = list(a = 1:10, b = 100, c = 1000),
         sub2 = list(animal1 = "ape", animal2 = "bear", animal3 = "cat"),
         sub3 = rep(c(TRUE, FALSE), 3))
js = toJSON(x)

fromJSON(js)
  # leave character strings uncollapsed
fromJSON(js, simplify = StrictNumeric + StrictLogical)
fromJSON(js, simplify = c(StrictNumeric, StrictLogical))


fromJSON(js, simplifyWithNames = FALSE)
fromJSON(js, simplifyWithNames = TRUE)

#######
#  stringFun
txt = '{ "magnitude": 3.8,
         "longitude": -125.012,
```

```
          "latitude": 40.382,
          "date":  "new Date(1335515917000)",
          "when": "/Date(1335515917000)/",
          "country": "USA",
          "verified": true
        }'

convertJSONDate =
function(x)
{
   if(grepl("/?(new )?Date\\(", x)) {
      val = gsub(".*Date\\(([0-9]+)\\).*", "\1", x)
      structure(as.numeric(val)/1000, class = c("POSIXct", "POSIXt"))
   } else
      x
}

fromJSON(txt, stringFun = convertJSONDate)

 #  A C routine for converting dates
jtxt = '[ 1, "/new Date(12312313)", "/Date(12312313)"]'
ans = fromJSON(jtxt)
ans = fromJSON(jtxt, stringFun = "R_json_dateStringOp")

 # A C routine that returns a char * - leaves strings as is
c = fromJSON(jtxt, stringFun = I("dummyStringOperation"))
c = fromJSON(jtxt, stringFun = I(getNativeSymbolInfo("dummyStringOperation")))
c = fromJSON(jtxt, stringFun =
                     I(getNativeSymbolInfo("dummyStringOperation")$address))

  # I() or class = "NativeStringRoutine".
c = fromJSON(jtxt, stringFun =
                       structure("dummyStringOperation",
                                 class = "NativeStringRoutine"))
```

`isValidJSON` *Test if JSON content is valid*

Description

This function and its methods allows the caller to verify if the JSON content is strictly valid. Even if the content is invalid, the parser may still be able to make sense of it or at least get it partially correct and yield a result. So this function allows the caller to verify that the input is legitimate and not just rely on the parser not failing.

Usage

```
isValidJSON(content, asText = inherits(content, "AsIs"), ...)
```

Arguments

content	the JSON input either as a string, the name of a file or URL, or a connection object.
asText	a logical value that specifies whether the value in content is actually the JSON content or the name of a file
...	additional parameters for the methods

Value

A logical value indicating whether the content is valid JSON (TRUE) or invalid (FALSE).

Author(s)

Duncan Temple Lang. Functionality suggested by Jeroen Ooms.

References

libjson

See Also

fromJSON

Examples

```
isValidJSON(I('{"foo" : "bar"}'))

isValidJSON(I('{foo : "bar"}'))
isValidJSON('{foo : "bar"}', TRUE)
```

---

readJSONStream *Read JSON from a Connection/Stream*

Description

This function is capable of reading and processing JSON content from a "stream". This is most likely to be from an R connection, but can be an arbitrary source of JSON content. The idea is that the parser will pull partial data from the source and process it immediately, and then return to retrieve more data. This allows the parser to work on the JSON content without it all being in memory at one time. This can save a significant amount of memory and make some computations feasible which would not be if we had to first read all of the JSON and then process it.

Usage

```
readJSONStream(con, cb = NULL, simplify = Strict, nullValue = NULL,
                 simplifyWithNames = TRUE)
```

Arguments

con	a connection object from which we will read the JSON content. This can also be any R expression that returns a string. This allows a caller to get content from any source, not just a connection.
cb	an optional callback function that is invoked for each top-level JSON object in the stream. Typically there will only be one such top-level object and so the callback is not really needed as the default is to return that top-level object from readJSONStream. However, if there are multiple top-level JSON objects in the stream, this callback function can process them, e.g. merge them, collapse the contents.
simplify	same as for fromJSON.
nullValue	same as for fromJSON.
simplifyWithNames	same as for fromJSON.

Value

By default, this returns the top-level JSON object in the stream.

Author(s)

Duncan Temple Lang

References

libjson and the JSONSTREAM facilities.

See Also

fromJSON and its methods, specifically the method for a connection.

Examples

```
## Not run:
xx = '[1,2, 3]{"a": [true, false]}'
con = textConnection(xx)

f = function(x)
       print(sum(unlist(x)))

readJSONStream(con, f)

  # The callback function can be anonymous
con = textConnection(xx)
readJSONStream(con, function(x)
                      print(sum(unlist(x))))
```

```
gen =
function() {
 ans <- 0
 list(update = function(x) ans <<- ans + sum(unlist(x)),
      value = function() ans)
}
g = gen()
con = textConnection(xx)
readJSONStream(con, g$update)

## End(Not run)
```

---

toJSON *Convert an R object to a string in Javascript Object Notation*

Description

This function and its methods convert an R object into a string that represents the object in Javascript Object Notation (JSON).

The different methods try to map R's vectors to JSON arrays and associative arrays. There is ambiguity here as an R vector of length 1 can be a JSON scalar or an array with one element. When there are names on the R vector, the descision is clearer. We have introduced the `emptyNamedList` variable to identify an empty list that has an empty names character vector and so maps to an associative array in JSON, albeit an empty one.

Objects of class `AsIs` in R, i.e. that are enclosed in a call to `I()` are treated as containers even if they are of length 1. This allows callers to indicate the desired representation of an R "scalar" as an array of length 1 in JSON

Usage

```
toJSON(x, container = isContainer(x, asIs, .level),
        collapse = "\n", ..., .level = 1L,
         .withNames = length(x) > 0 && length(names(x)) > 0, .na = "null",
         .escapeEscapes = TRUE, pretty = FALSE, asIs = NA, .inf = " Infinity")
```

Arguments

`x`	the R object to be converted to JSON format
`...`	additional arguments controlling the formatting of the JSON.
`container`	a logical value indicating whether to treat the object as a vector/container or a scalar and so represent it as an array or primitive in JavaScript.
`collapse`	a string that is used as the separator when combining the individual lines of the generated JSON content
`.level`	an integer value. This is not a parameter the caller is supposed to supply. It is a value that is passed in recursive calls to identify the top-level and sub-level serialization to JSON and so help to identify when a scalar needs to be in a container and when it is legitimate to output a scalar value directly.

.withNames	a logical value. If we are dealing with a named vector/list, we typically generate a JSON associative array/dictionary. If there are no names, we create a simple array. This argument allows us to explicitly control whether we use a dictionary or to ignore the names and use an array.
.na	a value to use when we encounter an NA value in the R objects. This allows the caller to convert these to whatever makes sense to them. For example, we might specify this as "null" and then the NA values will appear as null in the JSON output. One can also specify an unusual numeric value, e.g. -9999999 to indicate a missing value!
.escapeEscapes	a logical value that controls how new line and tab characters are serialized. If this is TRUE, we preserve them symbolically by escaping the \. Otherwise, we replace them with their literal value.
pretty	a logical value that controls if extra processing is done on the result to make it indented for easier human-readability. At present, this reparses the generated JSON content and re-formats it (using libjson). This means that there can be three copies of the data in memory simultaneously - the original data, the JSON text and the pretty-printed version of the JSON text. For large objects, this can require a lot of memory.
asIs	a logical value that, if TRUE causes R vectors of length 1 to be represented as arrays in JSON, but if FALSE to be represented as scalars, where appropriate (i.e. not the top level of the JSON content). This avoids having to explicitly mark sub-elements in an R object as being of class AsIs.
.inf	how to represent infinity in JSON. This should be a string.

Value

A string containing the JSON content.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.json.org

See Also

fromJSON

Examples

```
toJSON(1:10)
toJSON(rnorm(3))
toJSON(rnorm(3), digits = 4)

toJSON(c("Duncan", "Temple Lang"))

toJSON(c(FALSE, FALSE, TRUE))
```

```
  # List of elements
toJSON(list(1L, c("a", "b"), c(FALSE, FALSE, TRUE), rnorm(3)))
  # with digits controlling formatting of sub-elements
toJSON(list(1L, c("a", "b"), c(FALSE, FALSE, TRUE), rnorm(3)),
        digits = 10)

  # nested lists
toJSON(list(1L, c("a", "b"), list(c(FALSE, FALSE, TRUE), rnorm(3))))

  # with names
toJSON(list(a = 1L, c("a", "b"), c(FALSE, FALSE, TRUE), rnorm(3)))

setClass("TEMP", representation(a = "integer", xyz = "logical"))
setClass("TEMP1", representation(one = "integer", two = "TEMP"))

new("TEMP1", one = 1:10, two = new("TEMP", a = 4L, xyz = c(TRUE, FALSE)))


toJSON(list())
toJSON(emptyNamedList)
toJSON(I(list("hi")))
toJSON(I("hi"))


x = list(list(),
         emptyNamedList,
         I(list("hi")),
         "hi",
         I("hi"))
toJSON(x)

 # examples of specifying .withNames
toJSON(structure(1:3, names = letters[1:3]))
toJSON(structure(1:3, names = letters[1:3]), .withNames = FALSE)


 # Controlling NAs and mapping them to whatever we want.
toJSON(c(1L, 2L, NA), .na = "null")
toJSON(c(1L, 2L, NA), .na = -9999)

toJSON(c(1, 2, pi, NA), .na = "null")

toJSON(c(TRUE, FALSE, NA), .na = "null")

toJSON(c("A", "BCD", NA), .na = "null")

toJSON( factor(c("A", "B", "A", NA, "A")), .na = "null" )

toJSON(list(TRUE, list(1, NA), NA), .na = "null")
```

```
setClass("Foo", representation(a = "integer", b = "character"))
obj = new("Foo", a = c(1L, 2L, NA, 4L), b = c("abc", NA, "def"))
toJSON(obj)
toJSON(obj, .na = "null")

 # hexmode example with .na ?

toJSON(matrix(c(1, 2, NA, 4), 2, 2), .na = "null")
toJSON(matrix(c(1, 2, NA, 4), 2, 2), .na = -9999999)


x = '"foo\tbar\n\tagain"'
cat(toJSON(x))
cat(toJSON(list(x)))

 # if we want to expand the new lines and tab characters
cat(toJSON(x), .escapeEscapes = FALSE)


 # illustration of the asIs argument
 cat(toJSON(list(a = 1, b = 2L, c = TRUE,
                 d = c(1, 3),
                 e = "abc"), asIs = TRUE))

 cat(toJSON(list(a = 1, b = 2L, c = TRUE,
                 d = c(1, 3),
                 e = "abc"), asIs = FALSE))

  # extra level
 cat(toJSON(list(a = c(x = 1), b = 2L, c = TRUE,
                 d = list(1, 3),
                 e = "abc"), asIs = FALSE, pretty = TRUE))


  # data frame by row as arrays
 twoRows = data.frame(a = 1:2, b = as.numeric(1:2))
 j = toJSON(twoRows, byrow = TRUE)
 r = data.frame(do.call(rbind, fromJSON(j)))

  # here we keep the names of the columns on each row
  # which allows us to round-trip the object back to R
 j = toJSON(twoRows, byrow = TRUE, colNames = TRUE)
 r = data.frame(do.call(rbind, fromJSON(j)))
```

Index