Package 'V8'

March 3, 2016

Type Package

Title Embedded JavaScript Engine

Version 1.0.0

Author Jeroen Ooms

Maintainer Jeroen Ooms <jeroen.ooms@stat.ucla.edu>

Description An R interface to Google's open source JavaScript engine.
V8 is written in C++ and implements ECMAScript as specified in ECMA-262,
5th edition. In addition, this package implements typed arrays as
specified in ECMA 6 used for high-performance computing and libraries
compiled with 'emscripten'.

License MIT + file LICENSE

URL https://github.com/jeroenooms/v8, https://code.google.com/p/v8/

BugReports https://github.com/jeroenooms/v8/issues

SystemRequirements V8: libv8-dev (deb), v8-devel (rpm), v8-3.14 (arch)

NeedsCompilation yes

VignetteBuilder knitr

Imports Rcpp (>= 0.12), jsonlite (>= 0.9.14), curl (>= 0.5), utils

LinkingTo Rcpp

Suggests testthat, knitr, rmarkdown

RoxygenNote 5.0.1

Repository CRAN

Date/Publication 2016-03-03 14:05:42

R topics documented:

Context *Run JavaScript in a V8 context*

Description

A *context* is an execution environment that allows separate, unrelated, JavaScript code to run in a single instance of V8, like a tab in a browser.

Usage

```
v8(global = "global", console = TRUE, typed_arrays = TRUE)

new_context(global = "global", console = TRUE, typed_arrays = TRUE)
```

Arguments

global	character vector indicating name(s) of the global environment. Use NULL for no name.
console	expose console API (console.log, console.warn, console.error).
typed_arrays	enable support for typed arrays (part of ECMA6). This adds a bunch of additional functions to the global namespace.

Details

The v8 function is an alias for new_context, they do exactly the same thing.

V8 contexts cannot be serialized but creating a new contexts and sourcing code is very cheap. You can run as many parallel v8 contexts as you want. R packages that use V8 can use a separate V8 context for each object or function call.

The ct$eval method evaluates a string of raw code in the same way as eval would do in JavaScript. It returns a string with console output. The ctget, ctassign and ct$call functions on the other hand automatically convert arguments and return value from/to JSON, unless an argument has been wrapped in JS(), see examples. The ct$validate function is used to test if a piece of code is valid JavaScript syntax within the context, and always returns TRUE or FALSE.

JSON is used for all data interchange beteen R and JavaScript. Therefore you can (and should) only exchange data types that have a sensible JSON representation. All aguments and objects are automatically converted according to the mapping described in Ooms (2014), and implemented by the jsonlite package in fromJSON and toJSON.

The name of the global object (i.e. global in node and window in browsers) can be set with the global argument. A context always have a global scope, even when no name is set. When a context is initiated with global = NULL, the global environment can be reached by evaluating this in the global scope, for example: ct$eval("Object.keys(this)").

Methods

`console()` starts an interactive console

`eval(src)` evaluates a string with JavaScript source code

`validate(src)` test if a string of JavaScript code is syntactically valid

`source(file)` evaluates a file with JavaScript code

`get(name)` convert a JavaScript to R via JSON

`assign(name, value)` copy an R object to JavaScript via JSON

`call(fun, ...)` call a JavaScript function with arguments `...`. Arguments which are not wrapped in `JS()` automatically get converted to JSON

`reset()` resets the context (removes all objects)

References

A Mapping Between JSON Data and R Objects (Ooms, 2014): http://arxiv.org/abs/1403.2805

Examples

```
# Create a new context
ctx <- v8();

# Evaluate some code
ctx$eval("var foo = 123")
ctx$eval("var bar = 456")
ctx$eval("foo+bar")

# Functions and closures
ctx$eval("JSON.stringify({x:Math.random()})")
ctx$eval("(function(x){return x+1;})(123)")

# Objects (via JSON only)
ctx$assign("mydata", mtcars)
ctx$get("mydata")

# Assign JavaScript
ctx$assign("foo", JS("function(x){return x*x}"))
ctx$assign("bar", JS("foo(9)"))
ctx$get("bar")

# Validate script without evaluating
ctx$validate("function foo(x){2*x}") #TRUE
ctx$validate("foo = function(x){2*x}") #TRUE
ctx$validate("function(x){2*x}") #FALSE

# Use a JavaScript library
ctx$source(system.file("js/underscore.js", package="V8"))
ctx$call("_.filter", mtcars, JS("function(x){return x.mpg < 15}"))

# Example from underscore manual
```

```
ctx$eval("_.templateSettings = {interpolate: /\\{\\{(.+?)\\}\\}/g}")
ctx$eval("var template = _.template('Hello {{ name }}!')")
ctx$call("template", list(name = "Mustache"))

# Call anonymous function
ctx$call("function(x, y){return x * y}", 123, 3)

## Not run: CoffeeScript
ct2 <- new_context()
ct2$source("http://coffeescript.org/extras/coffee-script.js")
jscode <- ct2$call("CoffeeScript.compile", "square = (x) -> x * x", list(bare = TRUE))
ct2$eval(jscode)
ct2$call("square", 9)

# Interactive console
ct3 <- new_context()
ct3$console()
//this is JavaScript
var test = [1,2,3]
JSON.stringify(test)
exit
## End(Not run)
```

JS *Mark character strings as literal JavaScript code*

Description

This function `JS()` marks character vectors with a special class, so that it will be treated as literal JavaScript code. It was copied from the htmlwidgets package, and does exactly the same thing.

Usage

```
JS(...)
```

Arguments

`...`	character vectors as the JavaScript source code (all arguments will be pasted into one character string)

Author(s)

Yihui Xie

Examples

```
ct <- new_context()
ct$eval("1+1")
ct$eval(JS("1+1"))
ct$assign("test", JS("2+3"))
ct$get("test")
```

Index