Introduction to V8 for R

2020-06-18

V8 is Google's open source, high performance JavaScript engine. It is written in C++ and implements ECMAScript as specified in ECMA-262, 5th edition. The V8 R package builds on the C++ library to provide a completely standalone JavaScript engine within R:

```
# Create a new context
ct <- v8()

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

```
[1] "579"
```

A major advantage over the other foreign language interfaces is that V8 requires no compilers, external executables or other run-time dependencies. The entire engine is contained within a 6MB package (2MB zipped) and works on all major platforms.

```
# Create some JSON
cat(ct$eval("JSON.stringify({x:Math.random()})"))
```

```
{"x":0.5580623043314792}
```

```
# Simple closure
ct$eval("(function(x){return x+1;})(123)")
```

```
[1] "124"
```

However note that V8 by itself is just the naked JavaScript engine. Currently, there is no DOM (i.e. no *window* object), no network or disk IO, not even an event loop. Which is fine because we already have all of those in R. In this sense V8 resembles other foreign language interfaces such as Rcpp or rJava, but then for JavaScript.

Loading JavaScript Libraries

The `ct$source` method is a convenience function for loading JavaScript libraries from a file or url.

```
ct$source(system.file("js/underscore.js", package="V8"))
ct$source("https://cdnjs.cloudflare.com/ajax/libs/crossfilter/1.3.11/crossfilter.min.js")
```

Data Interchange

By default all data interchange between R and JavaScript happens via JSON using the bidirectional mapping implemented in the jsonlite (http://arxiv.org/abs/1403.2805) package.

```
ct$assign("mydata", mtcars)
ct$get("mydata")
```

- Loading JavaScript Libraries
- Data Interchange
- Function Calls
- Interactive JavaScript Console
- warnings, errors and console.log
- The Global Namespace
- Syntax Validation
- Callback To R

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger  15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9         27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2     26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa      30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L    15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino      19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E        21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Alternatively use `JS()` to assign the value of a JavaScript expression (without converting to JSON):

```
ct$assign("foo", JS("function(x){return x*x}"))
ct$assign("bar", JS("foo(9)"))
ct$get("bar")
```

```
[1] 81
```

Function Calls

The `ct$call` method calls a JavaScript function, automatically converting objects (arguments and return value) between R and JavaScript:

```
ct$call("_.filter", mtcars, JS("function(x){return x.mpg < 15}"))
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
```

It looks a bit like `.Call` but then for JavaScript instead of C.

Interactive JavaScript Console

A fun way to learn JavaScript or debug a session is by entering the interactive console:

```
# Load some data
data(diamonds, package = "ggplot2")
ct$assign("diamonds", diamonds)
ct$console()
```

From here you can interactively work in JavaScript without typing `ct$eval` every time:

```
var cf = crossfilter(diamonds)
var price = cf.dimension(function(x){return x.price})
var depth = cf.dimension(function(x){return x.depth})
price.filter([2000, 3000])
output = depth.top(10)
```

To exit the console, either press `ESC` or type `exit`. Afterwards you can retrieve the objects back into R:

```
output <- ct$get("output")
print(output)
```

warnings, errors and console.log

Evaluating invalid JavaScript code results in a SyntaxError:

```
# A common typo
ct$eval('var foo <- 123;')
```

```
Error in context_eval(join(src), private$context, serialize): SyntaxErro
r: Unexpected token '<'
```

JavaScript runtime exceptions are automatically propagated into R errors:

```
# Runtime errors
ct$eval("123 + doesnotexit")
```

```
Error in context_eval(join(src), private$context, serialize): ReferenceEr
ror: doesnotexit is not defined
```

Within JavaScript we can also call back to the R console manually using `console.log`, `console.warn` and `console.error`. This allows for explicitly generating output, warnings or errors from within a JavaScript application.

```
ct$eval('console.log("this is a message")')
```

```
this is a message
```

```
ct$eval('console.warn("Heads up!")')
```

```
Warning: Heads up!
```

```
ct$eval('console.error("Oh no! An error!")')
```

```
Error in context_eval(join(src), private$context, serialize): Oh no! An e
rror!
```

A example of using `console.error` is to verify that external resources were loaded:

```
ct <- v8()
ct$source("https://cdnjs.cloudflare.com/ajax/libs/crossfilter/1.3.11/crossfilter.min.js")
ct$eval('var cf = crossfilter || console.error("failed to load crossfilter!")')
```

The Global Namespace

Unlike what you might be used to from Node or your browser, the global namespace for a new context if very minimal. By default it contains only a few objects: `global` (a reference to itself), `console` (for `console.log` and friends) and `print` (an alias of console.log needed by some JavaScript libraries)

```
ct <- v8(typed_arrays = FALSE);
ct$get(JS("Object.keys(global)"))
```

```
[1] "print"   "console" "global"
```

If typed arrays are enabled it contains some additional functions:

```
ct <- v8(typed_arrays = TRUE);
ct$get(JS("Object.keys(global)"))
```

```
[1] "print"   "console" "global"
```

A context always has a global scope, even when no name is set. When a context is initiated with `global = NULL`, it can still be reached by evaluating the `this` keyword within the global scope:

```
ct2 <- v8(global = NULL, console = FALSE)
ct2$get(JS("Object.keys(this).length"))
```

```
[1] 1
```

```
ct2$assign("cars", cars)
ct2$eval("var foo = 123")
ct2$eval("function test(x){x+1}")
ct2$get(JS("Object.keys(this).length"))
```

```
[1] 4
```

```
ct2$get(JS("Object.keys(this)"))
```

```
[1] "print" "cars"  "foo"   "test"
```

To create your own global you could use something like:

```
ct2$eval("var __global__ = this")
ct2$eval("(function(){var bar = [1,2,3,4]; __global__.bar = bar; })()")
ct2$get("bar")
```

```
[1] 1 2 3 4
```

Syntax Validation

V8 also allows for validating JavaScript syntax, without actually evaluating it.

```
ct$validate("function foo(x){2*x}")
```

```
[1] TRUE
```

```
ct$validate("foo = function(x){2*x}")
```

```
[1] TRUE
```

This might be useful for all those R libraries that generate browser graphics via templated JavaScript. Note that JavaScript does not allow for defining anonymous functions in the global scope:

```
ct$validate("function(x){2*x}")
```

```
[1] FALSE
```

To check if an anonymous function is syntactically valid, prefix it with `!` or wrap in `()`. These are OK:

```
ct$validate("(function(x){2*x})")
```

```
[1] TRUE
```

```
ct$validate("!function(x){2*x}")
```

```
[1] TRUE
```

Callback To R

A recently added feature is to interact with R from within JavaScript using the `console.r API`. This is most easily demonstrated via the interactive console.

```
ctx <- v8()
ctx$console()
```

From JavaScript we can read/write R objects via `console.r.get` and `console.r.assign`. The final argument is an optional list specifying arguments passed to `toJSON` or `fromJSON`.

```
// read the iris object into JS
var iris = console.r.get("iris")
var iris_col = console.r.get("iris", {dataframe : "col"})

//write an object back to the R session
console.r.assign("iris2", iris)
console.r.assign("iris3", iris, {simplifyVector : false})
```

To call R functions use `console.r.call`. The first argument should be a string which evaluates to a function. The second argument contains a list of arguments passed to the function, similar to `do.call` in R. Both named and unnamed lists are supported. The return object is returned to JavaScript via JSON.

```
//calls rnorm(n=2, mean=10, sd=5)
var out = console.r.call('rnorm', {n: 2,mean:10, sd:5})
var out = console.r.call('rnorm', [2, 20, 5])

//anonymous function
var out = console.r.call('function(x){x^2}', {x:12})
```

There is also an `console.r.eval` function, which evaluates some code. It takes only a single argument (the string to evaluate) and does not return anything. Output is printed to the console.

```
console.r.eval('sessionInfo()')
```

Besides automatically converting objects, V8 also propagates exceptions between R, C++ and JavaScript up and down the stack. Hence you can catch R errors as JavaScript exceptions when calling an R function from JavaScript or vice versa. If nothing gets caught, exceptions bubble all the way up as R errors in your top-level R session.

```
//raise an error in R
console.r.call('stop("ouch!")')

//catch error from JavaScript
try {
  console.r.call('stop("ouch!")')
} catch (e) {
  console.log("Uhoh R had an error: " + e)
}
//# Uhoh R had an error: ouch!
```