John Coene

R and JavaScript

To my son,

without whom I should have finished this book two years earlier

Contents

List of Tables

List of Figures

Preface

The R programming language has seen the integration of many programming languages; C, C++, Python, to name a few, can be seamlessly embedded into R code. Little known to many, R works just as well with JavaScript—this book delves into the various ways both languages can work together.

The ultimate aim of this work is to put the reader at ease with inviting JavaScript in their data science workflow.

About the Author

Frida Gomam is a famous lady. Police will always let her go.

1

Introduction

In this chapter, after briefly going through prerequisites, we provide a rationale for integrating JavaScript with R which we support with examples, namely packages available on CRAN. Then, we list the various ways in which one might go about making both languages work together. Finally, we end with a review of concepts fundamental to fully understand the more advanced topics residing in the forthcoming chapters.

Prerequisites

The code contained in the following pages is approachable to readers with basic knowledge of R, although familiarity with package development using devtools[1] (Wickham et al., 2019), as well as the Shiny[2] (Chang et al., 2019) framework is helpful. The reason for the former is that some of the ways one builds integrations with JavaScript naturally take the form of packages. The following section thus runs over the essentials of building a package to ensure everyone can keep up. As both Shiny and JavaScript run in the browser they make for axiomatic companions; we'll therefore use Shiny extensively.

Only basic knowledge of JavaScript is required to understand and learn from the book as not only is JavaScript rather uncomplicated and its syntax similar to R's in places, we write surprisingly little of it. Understanding of JSON and HTML, however, is essential. In essence, if one has already used external JavaScript libraries in HTML or R markdown documents then one is well-equipped to tackle this book but in the event that you have not, we will go through a quick review of the basics of both JavaScript and JSON.

It is highly recommended to use the freely available RStudio IDE[3] to follow along, particularly if you are beginner in R.

[1] https://devtools.r-lib.org/
[2] https://shiny.rstudio.com/
[3] https://rstudio.com/products/rstudio/

Package Development

Developing packages used to be notoriously difficult but things have greatly changed namely thanks the devtools and more recent usethis[4] (Wickham and Bryan, 2019) packages. The first is specifically designed to help you with creating packages; setting up tests, running checks, building and installing packages, etc. The second, usethis, more broadly helps setting up your project, and automate repetitive tasks. Here, we only skim over the fundamentals, if you want to read more about it look for Hadley Wickham's R Packages[5] which is freely available on the web.

Start by installing the two packages from CRAN.

```
install.packages("devtools")
install.packages("usethis")
```

We'll create an admittedly ridiculous package to explain how it works. despite being useless, this package will introduce certain concepts of package development that are essential to understand for what is to come. The package will be named "test" and will provide a single function that will print a JavaScript file stored inside our package. Useless but bear with it.

Let's create the package, you can either do so using the RStudio IDE or with usethis. From the RStudio IDE go to `File > New Project > New Directory` then select "R package" and fill in the small form. But it could be argued that it's actually easier with usethis; type the code below in your R console to create a package named "test" in your current directory. If you run it from RStudio a new project window should open.

```
usethis::create_package("test")
```

This creates an empty package. Let us add the JavaScript file that our yet-to-be-written R function will eventually print out. R packages follow strict guidelines so this file cannot be store anywhere we want; it must be stored in a directory called `inst` which stands for "installed files." Create a directory named "inst" and place a file called "javascript.js" within it.

We'll create the following, extremely simple JavaScript file that contains code which declares a variable `x`.

```
// inst/javascript.js
var x = 3;
```

[4] https://usethis.r-lib.org/
[5] http://r-pkgs.had.co.nz/

We cab actually create this file from R with:

```
dir.create("inst") # create directory
writeLines("var x = 3;", con = "inst/javascript.js") # create file
```

Your current directory should now look something like this.

```
DESCRIPTION
NAMESPACE
R/
inst/
|-- javascript.js
```

Now we need to create the R function that will read and print this file. Then again, R is strict, your R files must be placed in the R directory. Below we create the file from the R console.

```
file.create("R/function.R")
```

In the `function.R` file we just created we place a function that reads the `javascript.js` file and prints it to the R console along with a simple message. To read the file R comes with a `system.file` function which will return the path the file. Indeed as the package is to be installed on different one can never use absolute paths, it will depend on where the user of your package has it installed. If curious, run `.libPaths()` to find where it is.

```
# R/function.R
print_file <- function(){
  file <- system.file("javascript.js", package = "test")
}
```

We can now add two lines, one read the `file` and another to print it.

```
print_file <- function(){
  file <- system.file("javascript.js", package = "test")
  content <- readLines(file)
  print(content)
}
```

Let's add a pretty message, using the cli[6] [R-cli] package, before we print the file as it allows introducing another concept; using external libraries in our package. To use cli we can run:

[6] https://cli.r-lib.org/

```
# install.packages("cli")
usethis::use_package("cli")
##   Adding 'cli' to Imports field in DESCRIPTION
##   Refer to functions with `cli::fun()`
```

It does exactly what it printed. The `DESCRIPTION` file includes information about the package and, very importantly, dependencies. The package "test" we are creating now depends on cli. You should now see the following entry in the `DESCRIPTION` file.

```
Imports:
    cli
```

We can add cli to our function, note that we use the namespace (or package name) followed by a double colon, as was printed by the `usethis::use_package` function we ran earlier.

```
print_file <- function(){
  file <- system.file("javascript.js", package = "test")
  cli::cli_alert_info("JavaScript code below.")
  content <- readLines(file)
  print(content)
}
```

It's not done just yet. Let's demonstrate why it is not, it allows introducing devtools. We can build our package with `devtools::install()`. After running it and typing `test::` in the console nothing appears. Odd. This is because we have not explicitly exported the `print_file` function. To do so we need another package called roxygen2 [R-roxygen2], it allows writing the documentation in the `.R` file rather than have to create separate file, a lifesaver. Using said documentation we can specify which function should be export with a "tag".

Tags are preceded by `#' @`, there are plenty of them, we don't explore any other here.

```
#' @export
print_file <- function(){
  file <- system.file("javascript.js", package = "test")
  cli::cli_alert_info("JavaScript code below.")
  content <- readLines(file)
  print(content)
}
```

Now one can run `devtools::document()` to document the package based on the tag we added: this exports the function. Then we can run `devtools::install()` to build and install the package.

```
test::print_file()
##  JavaScript code below.
## [1] "var x = 3;"
```

Finally, the cyclical nature of building packages substitute `devtools::install()` with `devtools::load_all()`, the latter does not install the package and simply loads all the functions and objects of the package in the environment, this is much faster than installing it.

1. Write some code
2. Run `devtools::document()`
3. Run `devtools::load_all()`
4. Repeat

JSON

JSON (JavaScript Object Notation) is a very popular data *interchange* format with which we will work extensively throughout this book, it is thus crucial that we have a good understanding of it before we plunge into the nitty-gritty. As one might foresee, if we want two languages to work together it is essential that we have a data format that can be understood by both—JSON lets us harmoniously pass data from one to the other. While it is natively supported in JavaScript, it can be graciously handled in R with the jsonlite package[7] (Ooms et al., 2018), in fact it is the serialiser used internally by all of the methods detailed in the previous section.

JSON is to all intents and purposes the equivalent of lists in R; a flexible data format that can store pretty much anything. Below we create a nested list and convert it to JSON with the help of jsonlite, we set `pretty` to `TRUE` to add indentation for clearer printing.

```
# install.packages("jsonlite")
library(jsonlite)

lst <- list(
  a = 1,
  b = list(
```

[7] https://CRAN.R-project.org/package=jsonlite

```
    c = c("A", "B")
  ),
  d = 1:5
)

toJSON(lst, pretty = TRUE)
```

```
## {
##   "a": [1],
##   "b": {
##     "c": ["A", "B"]
##   },
##   "d": [1, 2, 3, 4, 5]
## }
```

Looking closely at the list and JSON output above one quickly sees the resemblance. Something seems odd though, the first value in the list (`a = 1`) was serialised; to an array (vector) of length one (`"a": [1]`) where would probably expect an integer instead. This is not a mistake, we often forget that there are no scalar types in R and that `a` is in fact a vector as we can observe below.

```
x <- 1
is.vector(x)
```

```
## [1] TRUE
```

JavaScript, on the other hand, does have scalar types, more often than not we will want to convert our vectors of length one to scalar types rather than arrays of length one. To do so we need use the `auto_unbox` argument in `jsonlite::toJSON`, we'll do this most of the time we have to convert data to JSON.

```
toJSON(lst, pretty = TRUE, auto_unbox = TRUE)
```

```
## {
##   "a": 1,
##   "b": {
##     "c": ["A", "B"]
##   },
##   "d": [1, 2, 3, 4, 5]
## }
```

As demonstrated above the vector of length one was "unboxed" into an integer, with `auto_unbox` set to `TRUE` jsonlite will properly convert such vectors into

their appropriate type; integer, numeric, boolean, etc. If JSON is more or less the equivalent of lists in R one might wonder how jsonlite handles data frames.

```
# subset of built-in dataset
df <- cars[1:2, ]

toJSON(df, pretty = TRUE)
```

```
## [
##   {
##     "speed": 4,
##     "dist": 2
##   },
##   {
##     "speed": 4,
##     "dist": 10
##   }
## ]
```

What jsonlite does internally is essentially turn the data.frame into a list *rowwise* to produce a list for every row. This generally how rectangular data is represented in lists, for instance, `purrr::transpose` does the same. We can reproduce this with the snippet below, we remove row names and use apply to turn every row into a list.

```
row.names(df) <- NULL
df_list <- apply(df, 1, as.list)

toJSON(df_list, pretty = TRUE, auto_unbox = TRUE)
```

```
## [
##   {
##     "speed": 4,
##     "dist": 2
##   },
##   {
##     "speed": 4,
##     "dist": 10
##   }
## ]
```

Jsonlite of course also enables reading data from JSON into R with the function `fromJSON`.

```
json <- toJSON(df) # convert to JSON
fromJSON(json) # read from JSON
```

```
##   speed dist
## 1     4    2
## 2     4   10
```

It's important to note that jsonlite did the conversion back to a data frame. Therefore the code below also returns a data frame even though the object we initially converted to JSON is a list.

```
class(df_list)
```

```
## [1] "list"
```

```
json <- toJSON(df_list)
fromJSON(json)
```

```
##   speed dist
## 1     4    2
## 2     4   10
```

Jsonlite provides many more options and functions that will let you tune JSON data in read to and written from R. Also, the jsonlite package does far more than what we detailed but at this juncture this is an adequate understanding of things.

JavaScript

The book is not meant to teach one JavaScript, only to show how graciously it can work with R; in that endeavour we aim at writing little JavaScript so the book remains approachable to a wide audience. Let us just go through the very basics to ensure we know enough to get started with the next chapter.

In the event that you would want to try what we briefly explore here: the easiest way is to create an HTML file, write your code within a script tag and open the file in your web browser. The output can be observed in the console of the browser which can be opened with a right click and selecting "inspect" then going to the "console" tab.

```
 <!-- index.html -->
<html>
```

```
  <head>
  </head>
  <body>
    <p id="content">Trying JavaScript!</p>
  </body>
  <script>
    // place your JavaScript code here
  </script>
</html>
```

The first difference with R is that the end of every line should be marked with a semi-colon. JavaScript code will often work without but one should always include them to avoid issues. Below we use `console.log`, JavaScript equivalent of R's `print` function.

```
console.log("hello JavaScript") // bad
console.log("hello JavaScript"); // good
```

Another difference is that variables must be declared with keywords such as `var` or `const` to declare a constant.

```
x = 1; // bad
var x = 1; // good
```

One can declare a variable without assigning a value to it, to then do so later on.

```
// good
var y;
y = [1,2,3];
```

In R like in JavaScript, variables can be accessed from the parent environment (often referred to as "context" in the latter). One immense difference though is that while it is seen as bad practice in R it is not in JavaScript where it comes very useful.

```
# it works but don't do this
x <- 123
foo <- function(){
  print(x)
}
foo()
```

```
## [1] 123
```

The above R code can be re-written in JavaScript. Note the slight variation in the function declaration.

```
// this is perfectly fine
var x = 1;

function foo(){
  console.log(x);
}

foo()
```

One concept which does not exist in R is that of the "DOM" which stands for Document Object Model. When a web page is loaded, the browser creates a Document Object Model of the page which can be accessed in JavaScript from the `document` object. This lets the developer programmatically manipulate the page itself so one can for instance, remove an element, change the text of an element, and plenty more.

The JavaScript code below grabs the element where `id='content'` from the `document` with `getElementById` and replaces the text (`innerText`).

```
 <!-- index.html -->
<html>
  <head>
  </head>
  <body>
    <p id="content">Trying JavaScript!</p>
  </body>
  <script>
    var cnt = document.getElementById("content");
    cnt.innerText = "The text has changed";
  </script>
</html>
```

This of course only scratches the surface of JavaScript but provides ample understanding of the language to keep up with the next chapters.

Rationale

Why merge JavaScript and R? They are two fundamentally different languages that each have their strengths and weaknesses, combining the two allows making the most of their consolidated advantages and circumvent their respective limitations to produce software altogether better for it.

A fair reason to use JavaScript might simply be that the thing one wants to achieve in R has already been realised in JavaScript. Why reinvent the wheel when the solution already exists and that it can be made accessible from R? The R package lawn[8] (Chamberlain and Hollister, 2019) by Ropensci integrates turf.js[9], a brilliant library for geo-spatial analysis. JavaScript is not required to make those computations, it could be rewritten solely in R but that would be vastly more laborious than wrapping the JavaScript API in R.

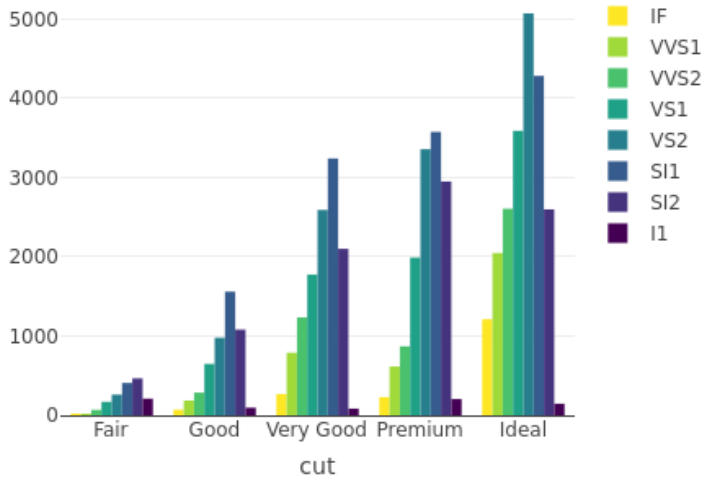
```
library(lawn)

lawn_count(lawn_data$polygons_count, lawn_data$points_count, "population")
```

```
## <FeatureCollection>
##   Bounding box: -112.1 46.6 -112.0 46.6
##   No. features: 2
##   No. points: 20
##   Properties:
##     values count
## 1 200, 600     2
## 2              0
```

Another great reason is that JavaScript can do things that R cannot, e.g.: run in the browser. Therefore one cannot natively create interactive visualisations with R. Plotly[10] (Sievert et al., 2019) by Carson Sievert packages the plotly JavaScript library[11] to let one create interactive visualisations solely from R code.

```
library(plotly)

plot_ly(diamonds, x = ~cut, color = ~clarity)
```

[8] https://github.com/ropensci/lawn
[9] http://turfjs.org/
[10] https://plotly-r.com/
[11] https://plot.ly/

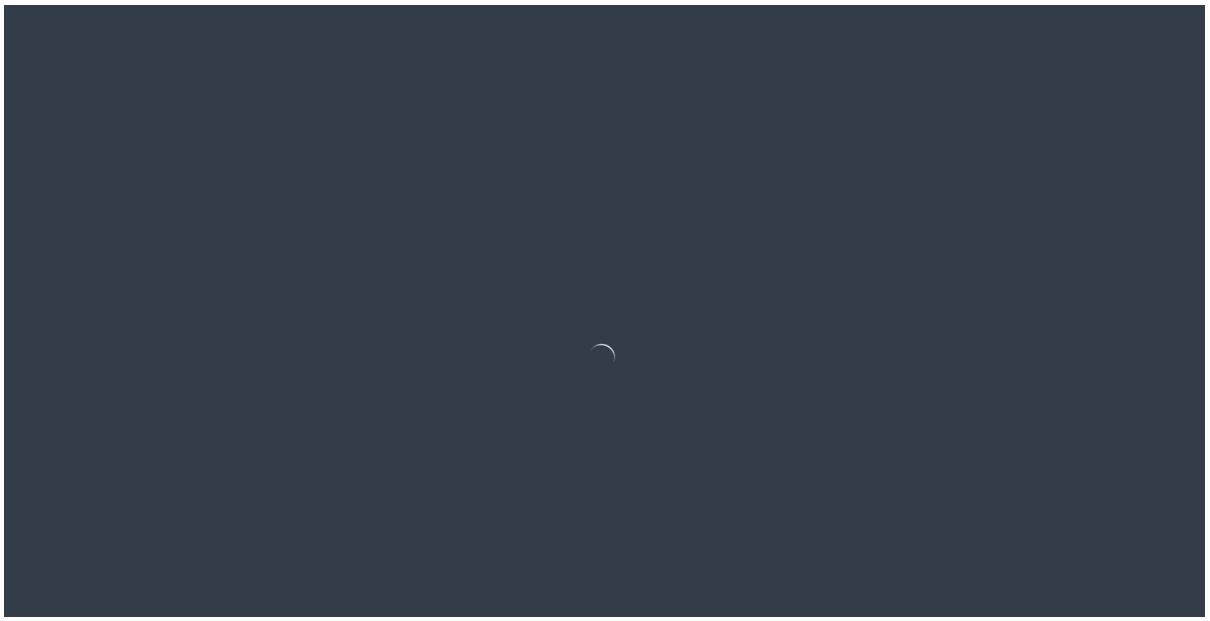


Finally, JavaScript can work together with R to improve how we communicate insights. One of the many ways in which Shiny stands out is that it lets one create web applications solely from R code with no knowledge of HTML, CSS, or JavaScript but that does not mean they can't extend Shiny, quite the contrary. The waiter package[12] (Coene, 2020) integrates a variety of JavaScript libraries to display loading screens in Shiny applications.

```
library(shiny)
library(waiter)

ui <- fluidPage(
  use_waiter(), # include dependencies
  actionButton("show", "Show loading for 3 seconds")
)

server <- function(input, output, session){
  # create a waiter
  w <- Waiter$new()

  # on button click
  observeEvent(input$show, {
```

[12] http://waiter.john-coene.com/

```
    w$show()
    Sys.sleep(3)
    w$hide()
  })
}

shinyApp(ui, server)
```



Hopefully this makes a couple of great reasons and alluring examples to entice you to persevere with this book.

Methods

Though perhaps not obvious at first, all of the packages used as examples in the previous section internally interface with JavaScript very differently. As we'll discover, there many ways in which one can blend JavaScript with R, generally the way to go about it is dictated by the nature of is to be achieved.

Let's list the methods available to us to blend JavaScript with R before covering them each in-depth in their own respective chapter later in the book.

V8

V8[13] by Jeroen Ooms is an R interface to Google's JavaScript engine. It will let you run JavaScript code directly from R and get the result back, it even comes with an interactive console. This is the way the lawn package used in a previous example has internally calls turf.js.

```
library(V8)

ctx <- v8()

ctx$eval("2 + 2") # this is evaluated in JavaScript!
```

```
## [1] "4"
```

htmlwidgets

htmlwidgets[14] (Vaidyanathan et al., 2019) specialises in wrapping JavaScript libraries that generate visual outputs. This is what packages such as plotly, DT[15] (Xie et al., 2019), highcharter[16] (Kunst, 2019), and many more use to provide interactive visualisation with R.

It is by far the most popular integration out there, at the time of writing this it has been downloaded nearly 10 million times from CRAN. It will therefore be covered extensively in later chapters.

Shiny

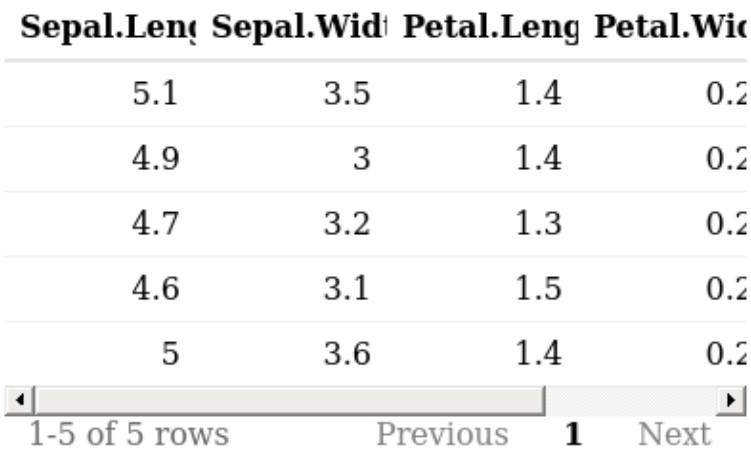
The Shiny framework allows creating applications accessible from web browsers where JavaScript natively runs, it follows that JavaScript can run *alongside* such applications. Often overlooked though, the two can also work *hand-in-hand* as one can pass data from the R server to the JavaScript front-end and vice versa. Some form of that tends to be included in htmlwidgets so one can pick up server-side which point on a scatter plot was clicked for instance.

[13] https://github.com/jeroen/v8
[14] http://www.htmlwidgets.org/
[15] https://rstudio.github.io/DT/
[16] http://jkunst.com/highcharter/

reactR

ReactR[17] (Inc et al., 2019) is an R package that emulates very well htmlwidgets but specifically for the React framework[18]. Unlike htmlwidgets it is not limited to visual outputs and also provides functions to build inputs, e.g.: a drop-down menu (like `shiny::selectInput`). The reactable package[19] (Lin, 2019) uses reactR to enable building interactive tables solely from R code.

```
reactable::reactable(iris[1:5, ], showPagination = TRUE)
```



Note that custom Shiny inputs can also be built, this is however not covered in this book for it is very well documented[20].

bubble

bubble[21] (Fay, 2020) by Colin Fay is a more recent R package, still under heady development but very promising: it lets one run node.js[22] code in R,

[17] https://react-r.github.io/reactR/
[18] https://reactjs.org/
[19] https://glin.github.io/reactable/
[20] https://shiny.rstudio.com/articles/building-inputs.html
[21] https://github.com/ColinFay/bubble
[22] https://nodejs.org/en/

comes with an interactive node REPL, the ability to install npm packages, and even an R markdown engine. It's similar to V8 in many ways.

```
library(bubble)

n <- NodeSession$new()

n$eval("2 + 2") # this is evaluated in node.js
```

```
## 4
```

r2d3

r2d3[23] (Luraschi and Allaire, 2018) by RStudio is an R package designed specifically to work with d3.js[24]. It is similar to htmlwidgets but works rather differently.

```
# https://rstudio.github.io/r2d3/articles/gallery/chord/
r2d3::r2d3(data = matrix(round(runif(16, 1, 10000)), ncol = 4, nrow = 4), script = "chord.
```

[23] https://rstudio.github.io/r2d3/
[24] https://d3js.org/

The packages bubble and V8 are intended for use of JavaScript for computations while r2d3, htmlwidgets, and reactR are designed to produce visual outputs (e.g.: graphs and tables), using JavaScript in Shiny can of course be used for the latter but is certainly not limited to that.

2

The V8 Engine

V8 is an R interface to Google's open source JavaScript engine of the same name, it powers Google Chrome, node.js and many other things. It is the first integration of JavaScript with R that we cover in this book. Both the V8 package and the engine it wraps are simple yet amazingly powerful.

Installation

First install the V8 engine itself, instructions to do so are well detailed on V8's README[1] and below.

On Debian or Ubuntu use the code below from the terminal to install libv8[2].

```
sudo apt-get install -y libv8-dev
```

On Centos install v8-devel, which requires the EPEL tools.

```
sudo yum install epel-release
sudo yum install v8-devel
```

On Mac OS use Homebrew[3].

```
brew install v8
```

Then install the R package from CRAN.

```
install.packages("V8")
```

[1] https://github.com/jeroen/v8#installation
[2] https://v8.dev/
[3] https://brew.sh/

Basics

V8 provides a reference class via R6[4] (Chang, 2019), which pertains to object-oriented programming, hence it might look unconventional to many R users. It's nonetheless easy to grasp. If one wants to learn more about the R6's reference class system Hadley Wickham has a very good chapter on it in his Advanced R[5] book.

Let's explore the basic functionalities of the package. First, load the library and use the function `v8` to instantiate a class.

```
library(V8)

engine <- v8()
```

The `eval` method allows running JavaScript code from R and retrive the results.

```
engine$eval("var x = 3 + 4;") # this is evealuated in R
engine$eval("x")
```

```
## [1] "7"
```

Two observations on the above snippet of code. First, the variable we got back in R is a character vector when it should have been either an integer or a numeric. This is because we used the `eval` method which merely prints the output, `get` is more appropriate; it converts it to an appropriate R equivalent.

```
# retrieve our previously created variable
(x <- engine$get("x"))
```

```
## [1] 7
```

```
class(x)
```

```
## [1] "integer"
```

Second, while creating a scalar with `eval("var x = 1;")` appears painless, imagine if you will the horror of having to convert a data frame to a JavaScript

[4] https://github.com/r-lib/R6
[5] https://adv-r.hadley.nz/r6.html

array via jsonlite then flatten it to character string so it can be used with the `eval` method. Horrid. Thankfully V8 comes with a method `assign`, complimentary to `get`, which declares R objects as JavaScript variables. It takes two arguments, first the name of the variable to create, second the object to assign to it.

```
# assign and retrieve a data.frame
engine$assign("vehicles", cars[1:3, ])
engine$get("vehicles")
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
```

All of the conversion is handled by V8 internally with jsonlite as demonstrated in the previous chapter. We can confirm that the data frame was converted to a list rowwise; using `JSON.stringify` to display how the object is store in V8.

```
cat(engine$eval("JSON.stringify(vehicles, null, 2);"))
```

```
## [
##   {
##     "speed": 4,
##     "dist": 2
##   },
##   {
##     "speed": 4,
##     "dist": 10
##   },
##   {
##     "speed": 7,
##     "dist": 4
##   }
## ]
```

However the cyclical loop of 1) creating a variable in JavaScript to 2) run a function on the aforementioned object 3) get the results back in R, can be tedious. So V8 also allows calling JavaScript functions on R objects directly with the `call` method and obtain the results back in R.

```
engine$eval("new Date();") # using eval
```

```
## [1] "Thu Jan 23 2020 21:02:05 GMT+0100 (CET)"
```

```
engine$call("Date", Sys.Date()) # using call
```

```
## [1] "Thu Jan 23 2020 21:02:05 GMT+0100 (CET)"
```

Finally, one can run code interactively rather than as strings by calling the console from the engine with `engine$console()` you can then exit the console by typing `exit` or hitting the ESC key.

External Libraries

V8 is actually quite bare in and of itself, there is for instance no functionalities built-in to read or write files from disk, it thus becomes truly interesting when you can use it JavaScript libraries. We do so with the `source` method which takes a `file` argument that will accept a path or URL to a JavaScript file to source. We'll demonstrate this using fuse.js[6] a fuzzy-search library.

```
engine$source("https://cdnjs.cloudflare.com/ajax/libs/fuse.js/3.4.6/fuse.min.js")
```

```
## [1] "true"
```

You can think of it as using the `script` tag in HTML to source (`src`) said file from disk or CDN.

```
<html>
  <head>
    <script src='https://cdnjs.cloudflare.com/ajax/libs/fuse.js/3.4.6/fuse.min.js'></scrip
  </head>
  <body>
    <p>Content</p>
  </body>
</html>
```

With the library imported we can use its functionalities. We'll replicate an example from fuse.js official website where it executes a search on an object that contains books and looks like JSON below.

[6] https://fusejs.io/

```
var books = [{
  'ISBN': 'A',
  'title': "Old Man's War",
  'author': 'John Scalzi'
}, {
  'ISBN': 'B',
  'title': 'The Lock Artist',
  'author': 'Steve Hamilton'
}]
```

This can be easily created, as we've already seen this is just how V8 creates data frames. We define a data.frame of books that looks similar and load it into our engine.

```
books <- data.frame(
  title = c(
    "Rights of Man",
    "Black Swan",
    "Common Sense",
    "Sense and Sensibility"
  ),
  id = c("a", "b", "c", "d")
)

engine$assign("books", books)
```

Then again we can make sure that the data.frame was turned into a rowwise JSON object.

```
cat(engine$eval("JSON.stringify(books, null, 2);"))
```

```
## [
##   {
##     "title": "Rights of Man",
##     "id": "a"
##   },
##   {
##     "title": "Black Swan",
##     "id": "b"
##   },
##   {
##     "title": "Common Sense",
##     "id": "c"
```

```
##   },
##   {
##     "title": "Sense and Sensibility",
##     "id": "d"
##   }
## ]
```

Now we can define options for our search, we don't get into the details of fuse.js here as this is not the purpose of this book, you can read more about the options in the examples section[7] of the site. We can mimic the format of the JSON options shown on the website with a simple list and assign that to our engine. If this confuses read the JSON section of the introduction[8].

```
// JavaScript
var options = {
  keys: ['title'],
  id: 'id'
}
```

```
# R
options <- list(
  keys = list("title"),
  id = "id"
)

engine$assign("options", options)
```

Then we can finish the second step of the online examples, instantiate a fuse.js object with our books and options objects then make a simple search, we assign the results of the search to an object which we then retrieve in R with `get`.

```
engine$eval("var fuse = new Fuse(books, options)")
engine$eval("var results = fuse.search('sense')")
engine$get("results")
```

```
## [1] "d" "c"
```

A search for "sense" returns a vector of ids where the term "sense" was found; `c` and `d` or the books Common Sense, Sense and Sensibility. We could perhaps make that last code simpler using the `call` method.

[7] https://fusejs.io/#Examples
[8] intro.html#json

```
engine$call("fuse.search", "sense")
```

```
## [1] "d" "c"
```

With Npm

We can also use npm[9] packages, though not all will work. Npm is Node's Package Manager, or in a sense Node's equivalent of CRAN.

To use npm packages we need browserify[10], a node library to bundle all dependencies of an npm package into a single, file which we can subsequently source in V8. Browserify is itself an npm package and there requires node and npm installed.

You can install browserify globally with the following command from the terminal.

```
npm install -g browserify
```

We can now browserify an npm package. To demonstrate we will use ms[11] which converts various time formats to milliseconds. First we install the package.

```
npm install ms
```

Then we browserify it. The first line creates a file called `in.js` which contains `global.ms = require('ms');` we then call browserify on that file specifying `ms.js` as output file.

```
echo "global.ms = require('ms');" > in.js
browserify in.js -o ms.js
```

We can now source `ms.js` with v8.

[9] https://www.npmjs.com/
[10] http://browserify.org/
[11] https://github.com/zeit/ms

```
library(V8)

ms <- v8()
ms$source("ms.js")
```

Then use the library.

```
ms$eval("ms('2 days')")
```

```
## [1] "172800000"
```

```
ms$eval("ms('1y')")
```

```
## [1] "31557600000"
```

Use in Packages

In this section we detail how one should go about using V8 in an R package, if you are not familiar with package development you can skip ahead.

Create a package however you usually do, using usethis, devtools or the RStudio IDE interface. Below we create a package called "ms" that will hold functionalities we explored in the previous section on npm packages.

```
R -e "usethis::create_package('ms')"
cd ./ms
```

The package is going to rely on V8 so you can add it under `Imports` in the `DESCRIPTION`. We are going to need to instantiate the class at some point (`engine <- v8()`).

One could perhaps require the user to do create such an object but it would not be convenient. Instead we can use `.onLoad`. You can read more about this function Hadley Wickham's Advanced R book[12]. The Python integration of R, reticulate[13] (Ushey et al., 2019) also advises this method[14] to import modules too. We often see this function placed in a `zzz.R` file.

[12] http://r-pkgs.had.co.nz/r.html
[13] https://rstudio.github.io/reticulate
[14] https://rstudio.github.io/reticulate/articles/package.html

```
# zzz.R
ms <- NULL

.onLoad <- function(libname, pkgname){
  ms <<- v8()
}
```

Our package should also include the external library `ms.js` we built from the npm package. We should place it in the `inst` directory. Create it and place the `ms.js` file within the latter.

```
mkdir -p inst
```

This should give a directory similar to this, for brevity we exclude `DESCRIPTION`, `NAMESPACE`, and other files that make up a package.

```
R/
| zzz.R
inst/
| -- ms.js
```

Now the dependency can be sourced in the `.onLoad` function. We can access the files in the `inst` directory with the `system.file` function. When using the `.onLoad` function it is goo practice to clean up with `.onUnload`.

```
# zzz.R
ms <- NULL

.onLoad <- function(libname, pkgname){
  ms <<- v8()

  dep <- system.file("ms.js", package = "ms")
  ms$source(dep)
}

.onUnload <- function(libpath){
  ms$reset()
}
```

We can then create a `to_ms` function, it will have access the `ms` object we instantiated in `.onLoad`.

```
#' Convert To Millisecond
#'
#' Convert to milliseconds to various formats.
#'
#' @param string String to convert.
#'
#' @export
to_ms <- function(string){
  ms$call("ms", string)
}
```

After running `devtools::document()` and `devtools::load_all()` the call `to_ms("2 days")` will return `172800000`.

3

Node.js with Bubble

A more recent R package, called bubble which will will let you run node.js code from R, the package comes with an REPL and an R markdown engine. As it is still under heavy development the package is not yet available on CRAN, you can get it from Github using either the devtools or remotes package[1] (Hester et al., 2019).

```
# install.packages("remotes")
remotes::install_github("ColinFay/bubble")
```

Basics

bubble is very similar to V8, it's also a reference class and the name of the methods are identical.

```
library(bubble)

n <- NodeSession$new()
n$eval("2 + 2;")
```

```
## 4
```

You can also assign and get variables, just like with v8.

```
n$assign(vehicles, cars[1:2, ])
n$get(vehicles)
```

```
##   speed dist
## 1     4    2
## 2     4   10
```

[1] https://remotes.r-lib.org/

Bubble also comes with an REPL terminal (read-eval-print loop), which can be launched with `bubble::node_repl()`.

R Markdown Engine

Bubble comes with an R markdown engine so JavaScript code can be evaluated in node from an R markdown document such as this one. To do so we simply need to place `bubble::set_node_engine()` at the top of the document, subsequent `node` chunks will be evaluated in a node session.

```
bubble::set_node_engine()
```

Once set we can run node code.

```
console.log(2 + 3);
```

```
## 5
## undefined
```

Npm

Npm is to node.js what CRAN is to R; a repository of packages that can be conveniently installed. Such packages can be installed using another Reference class called `NPM`. While with V8 one needs to "browserify" npm packages in hope that they work, with bubble, since it interacts with node js directly, there is no need for to "browserify" packages and we can be assured that they work.

Let us demonstrate with the natural[2] package which provides general natural language facility.

```
# initialise and install
npm <- Npm$new()$install("natural")
```

```
## > Add `node_modules` to .gitignore
```

[2] https://github.com/NaturalNode/natural

The above snippet initialises npm, which creates the pacakge.json file and a node_modules directory. The first is a DESCRIPTION file for node projects so to speak, the latter is a directory to hold dependencies installed.

The packages can then be imported in the node session and interacted with.

```
n$eval("const natural = require('natural')")
```

```
## undefined
```

```
n$eval("var tokenizer = new natural.WordTokenizer();")
```

```
## undefined
```

```
n$eval("tokenizer.tokenize('Using nodejs from R with npm.')")
```

```
## [ 'Using', 'nodejs', 'from', 'R', 'with', 'npm' ]
```

Use in Packages

To demonstrate how to use bubble in packages we shall build one to hold the functions we explored in the previous section. Then again, if you are not familiar with package development skip this section.

Create a package called "organic" using devtools, usethis, or the RStudio IDE. Then edit the DESCRIPTION file so that it imports the bubble package, since it is not yet on CRAN the Github dependency should be specified under `Remotes`.

```
Package: organic
Title: Natural Language Facilities
Imports:
  bubble
Remotes:
  ColinFay/bubble
```

Bibliography

Chamberlain, S. and Hollister, J. (2019). *lawn: Client for 'Turfjs' for 'Geospatial' Analysis*. R package version 0.5.0.

Chang, W. (2019). *R6: Encapsulated Classes with Reference Semantics*. R package version 2.4.1.

Chang, W., Cheng, J., Allaire, J., Xie, Y., and McPherson, J. (2019). *shiny: Web Application Framework for R*. R package version 1.4.0.

Coene, J. (2020). *waiter: Loading Screen for 'Shiny'*. R package version 0.1.1.9000.

Fay, C. (2020). *bubble: Launch and Interact with a NodeJS Session*. R package version 0.0.0.9003.

Hester, J., Csárdi, G., Wickham, H., Chang, W., Morgan, M., and Tenenbaum, D. (2019). *remotes: R Package Installation from Remote Repositories, Including 'GitHub'*. R package version 2.1.0.

Inc, F., Weststrate, M., Russell, K., and Dipert, A. (2019). *reactR: React Helpers*. R package version 0.4.1.

Kunst, J. (2019). *highcharter: A Wrapper for the 'Highcharts' Library*. R package version 0.7.0.

Lin, G. (2019). *reactable: Interactive Data Tables Based on 'React Table'*. R package version 0.1.0.

Luraschi, J. and Allaire, J. (2018). *r2d3: Interface to 'D3' Visualizations*. R package version 0.2.3.

Ooms, J., Temple Lang, D., and Hilaiel, L. (2018). *jsonlite: A Robust, High Performance JSON Parser and Generator for R*. R package version 1.6.

Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., and Despouy, P. (2019). *plotly: Create Interactive Web Graphics via 'plotly.js'*. R package version 4.9.1.

Ushey, K., Allaire, J., and Tang, Y. (2019). *reticulate: Interface to 'Python'*. R package version 1.13.

Vaidyanathan, R., Xie, Y., Allaire, J., Cheng, J., and Russell, K. (2019). *htmlwidgets: HTML Widgets for R*. R package version 1.5.1.

Wickham, H. and Bryan, J. (2019). *usethis: Automate Package and Project Setup*. R package version 1.5.1.

Wickham, H., Hester, J., and Chang, W. (2019). *devtools: Tools to Make Developing R Packages Easier*. R package version 2.2.1.

Xie, Y., Cheng, J., and Tan, X. (2019). *DT: A Wrapper of the JavaScript Library 'DataTables'*. R package version 0.10.