

John Coene

JavaScript for R

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 so one can easily call code written in other languages from the R console. 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. In that respect the book is not teaching one JavaScript but rather we show how little JavaScript can greatly support and enhance R code. Therefore focus is on integrating external JavaScript libraries and no prior knowledge of JavaScript is required.

Disclaimer

This book is currently a work in progress.





Part I

Basics & Roadmap



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 already 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.

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`¹ (?) by Ropensci integrates `turf.js`², 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
```

¹<https://github.com/ropensci/lawn>

²<http://turfjs.org/>

```
## 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. Plotly³ (?) by Carson Sievert packages the plotly JavaScript library⁴ to let one create interactive visualisations solely from R code.

```
library(plotly)

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

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

³<https://plotly-r.com/>

⁴<https://plot.ly/>

contrary. The waiter package⁵ (?) 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$show, {
    w$show()
    Sys.sleep(3)
    w$hide()
  })
}

shinyApp(ui, server)
```

⁵<http://waiter.john-coene.com/>



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

Prerequisites

The code contained in the following pages is approachable to readers with basic knowledge of R, although familiarity with package development using devtools⁶ (?), as well as the Shiny⁷ (?) framework is helpful. The reason for the former is that some of the ways one builds integrations with JavaScript naturally take the form of R 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

⁶<https://devtools.r-lib.org/>

⁷<https://shiny.rstudio.com/>

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 IDE⁸ to follow along, particularly if you are beginner in R.

Package Development

Developing packages used to be notoriously difficult but things have greatly changed, namely thanks to the devtools and more recent usethis⁹ (?) packages. The first is specifically designed to help creating packages; setting up tests, running checks, building and installing packages, etc. The second, usethis, more broadly helps setting up projects, and automating repetitive tasks. Here, we only skim over the fundamentals, there is an entire book by Hadley Wickham called R Packages¹⁰ solely dedicated to the topic.

Start by installing the two packages from CRAN.

```
install.packages(c("devtools", "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 following. 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, 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 called "test". Let's add the JavaScript file that the yet-to-be-written R function will eventually print out. R packages must follow strict rules, this file cannot be placed anywhere we want; it must be stored in a directory called **inst** which stands for "installation;" these are files that will simply be copied to the users' machine when they install your

⁸<https://rstudio.com/products/rstudio/>

⁹<https://usethis.r-lib.org/>

¹⁰<http://r-pkgs.had.co.nz/>

package. Create the `inst` directory in the root of the package (e.g.: next to the already existing `R`) and place a file called `javascript.js` within it.

Let's create the following, extremely simple JavaScript file that contains code which declares a variable named `x`.

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

This can all be done from the R console as shown below.

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

This should produce a directory looking something like:

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

Onto creating the R function that will read and print this file. Then again, R is strict, R files must be placed in the `R` directory. Then again, this can be done from the console or the RStudio IDE.

```
# 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. In order to read the file one needs to locate it, as packages are destined to be shared and installed on different machines one can never use absolute paths, where the `javascript.js` file is located will depend on where the user has installed the package: as the developer you can never know that for sure. Therefore R comes with a `system.file` function which will look for a file *relative to a package inst directory* and return its full path.

```
# R/function.R
print_file <- function(){
  # look for javascript.js in the inst folder of the test package
  file <- system.file("javascript.js", package = "test")
}
```

With the file path one can read the contents of the file and 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 cli¹¹ (?) package, before we print the file as it allows introducing another concept; using external libraries in our package. To use cli we can run:

```
# 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 we are creating will depend on the cli package; the DESCRIPTION file essentially tells R to check whether it is installed on the machine, if not it installs it (packages are for sharing). 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 allows introducing devtools. We can install the "test" package we created with devtools::install(), afterwards we can expect to be able to run the print_file function we wrote but it returns the following error: could not find function "print_file".

¹¹<https://cli.r-lib.org/>

Odd. This is because we have not explicitly “exported” the function, by default functions and objects declared in .R files (in packages) can only be used internally (within the package itself). In order to export it we need another package called roxygen2 (?), it allows writing the documentation in the .R file rather than have to create separate file, a lifesaver. Note that roxygen2 should already be installed on your machine. Using said documentation we can specify which function should be exported 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

Note the package we built is good enough for your own machine but won’t pass CRAN checks as we miss a lot of the documentation (roxygen2 tags).

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` package¹² (?), 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 but this is an argument you should omit when writing production code, it will reduce the file size (fewer spaces = smaller file size).

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

lst <- list(
  a = 1,
  b = list(
    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 one would probably expect an integer instead, `1` not `[1]`. 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.

¹²<https://CRAN.R-project.org/package=jsonlite>

```
x <- 1
length(x)
```

```
## [1] 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 dataframes.

```
# 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 sub-list for every row then it serialises to JSON. This is 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 how JSON data is read and written. Also, the jsonlite package does far more than what we detailed in this section 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 (e.g.: `index.html`), 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 future headaches. 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]; // define it as array  
y = 'string'; // change to character string
```

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 in R  
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 web 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, add an element (e.g.: a button), change the text of another, and plenty more.

The JavaScript code below grabs the element where `id='content'` from the document with `getElementById` and replaces the text (`innerText`). Even though our page only contains “Trying JavaScript!” when the page is opened (loaded) in your web browser JavaScript runs our code and changes it: this happens very fast so you likely don’t even see the original text.

```
<!-- 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. Also, a somewhat interesting fact that will prove useful later in the book: the RStudio IDE is actually a browser, therefore, in the IDE, one can right-click and “inspect element” to rendered source code.

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 are many ways in which one can blend JavaScript with R, generally the way to go about it is dictated by the nature of what 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

V8¹³ 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 internally calls turf.js.

```
library(V8)

## Using V8 engine 3.14.5.9

ctx <- v8()

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

## [1] "4"
```

htmlwidgets

htmlwidgets¹⁴ (?) specialises in wrapping JavaScript libraries that generate visual outputs. This is what packages such as plotly, DT¹⁵ (?), highcharter¹⁶ (?), 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

ReactR¹⁷ (?) is an R package that emulates very well `htmlwidgets` but specifically for the React framework¹⁸. 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` package¹⁹ (?) uses `reactR` to enable building interactive tables solely from R code.

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

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
5.1	3.5	1.4	0.2
4.9	3	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5	3.6	1.4	0.2

1-5 of 5 rows
Previous
1
Next

Note that custom Shiny inputs can also be built, this is however not covered in this book for it is very well documented²⁰.

bubble

`bubble`²¹ (?) by Colin Fay is a more recent R package, still under heavy development but very promising: it lets one run `node.js`²² code in R, comes

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/>

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()

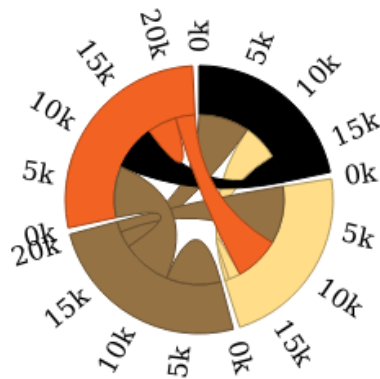
n$eval("2 + 2") # this is evaluated in node.js

## 4
```

r2d3

r2d3²³ (?) by RStudio is an R package designed specifically to work with d3.js²⁴. 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.

Part II

JavaScript for Computations



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 README¹ and below.

On Debian or Ubuntu use the code below from the terminal to install libv8².

```
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 Homebrew³.

```
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 R6⁴ (?), 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 R⁵ 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 retrieve the results.

```
engine$eval("var x = 3 + 4;") # this is evaluated in R
engine$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] "Wed Jun 03 2020 21:23:13 GMT+0200 (CEST)"
```

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

```
## [1] "Wed Jun 03 2020 21:23:13 GMT+0200 (CEST)"
```

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'></script>
  </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 section⁷ 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 introduction⁸.

```
// 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 npm⁹ 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 browserify¹⁰, 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 ms¹¹ 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 book¹². The Python integration of R, `reticulate`¹³ (?) also advises this method¹⁴ 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 good practice to clean up with `.onUnload`.

```
# zzz.R
ms <- NULL

.onLoad <- function(libname, pkgname){
  ms <- V8::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. Finally,, remember to add V8 as dependency.

```
usethis::use_package("V8")
```

3

Node.js with Bubble

A more recent R package, called bubble which 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 package¹ (?).

```
# 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 package.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.

Create a package called “organic” using devtools, the RStudio IDE, or usethis as shown below.

```
usethis::create_package("organic")
```

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
```





Part III

Web Development with Shiny



4

Shiny

Shiny is by far most popular, if not the only, web framework for the R programming language. In this chapter, after brushing up on the necessary to include JavaScript in shiny applications, we explore how to blend the JavaScript language with our R server and front-end.

Then again, the aim is not to write a lot of convoluted JavaScript, on the contrary, it is to write as little as possible and demonstrate to the reader that it is often enough to greatly improve the user experience of shiny applications. Making shiny work with JavaScript can essentially be broken down into two operations: 1) passing data from the R server to the JavaScript client and 2) the other way around, from the client to the R server.

In this chapter every section builds upon the previous, only skip those if you truly feel confident you understand it fully or you will get confused.

Static Files

In order to introduce JavaScript to shiny applications one must understand static files and how they work with the framework. Static files are files that are downloaded by the clients, in this case web browsers accessing shiny applications, as-is, these generally include images, CSS (`.css`), and JavaScript (`.js`).

If you are familiar with R package development, static files are to shiny applications what the “inst” directory is to an R package, those files are installed as-is and do not require further processing as opposed to the “src” folder which contains files that need compiling for instance.

There are numerous ways to run a shiny application locally; the two most used probably are `shinyApp` and `runApp`. The RStudio IDE comes with a convenient “Run” button when writing a shiny application, which when clicked actually uses the function `shiny::runApp` in the background, this function looks for said static files in the `www` directory and makes them available at the same path (`/www`). If you are building your applications outside of RStudio, you should

either also use `shiny::runApp` or specify the directory which then allows using `shiny::shinyApp`. Note that this only applies locally, shiny server (community and pro) as well as shinyapps.io¹ use the same defaults as the RStudio IDE and `shiny::runApp`.

In order to ensure the code in this book can run regardless of the reader's machine or editor, the asset directory is always specified explicitly. This is probably advised to steer clear of the potential headaches as, unlike the default, it'll work regardless of the environment. If you are using golem² (?) to develop your application then you should not worry about this as it specifies the directory internally.

Below we build a basic shiny application, however, before we define the ui and server we use the `shiny::addResourcePath` function to specify the location of the directory of static files that will be served by the server and thus accessible by the client. This function takes two arguments, first the `prefix`, which is the path (URL) at which the assets will be available, second the path to the directory of static assets.

We thus create the “assets” directory and a JavaScript file called `script.js` within it.

```
# run from root of app (where app.R is located)
dir.create("assets")
writeLines("console.log('Hello JS!');", con = "assets/script.js")
```

We can now use the `shiny::addResourcePath` to point to this directory. Generally the same name for the directory of static assets and prefix is used so as to avoid confusion, below we name them differently in order for the reader to clearly distinguish which is which.

```
# app.R
library(shiny)

# serve the files
addResourcePath(
  # will be accessible at /files
  prefix = "files",
  # path to our assets directory
  directoryPath = "assets"
)

ui <- fluidPage(
```

¹<https://www.shinyapps.io/>

²<https://thinkr-open.github.io/golem/>

```
  h1("R and JavaScript")
)

server <- function(input, output){}

shinyApp(ui, server)
```

If you then run the application and open it at the `/files/script.js` path (e.g.: `127.0.0.1:3000/files/script.js`) you should see the content of our JavaScript file (`console.log('Hello JS!')`), commenting the `addResourcePath` line will have a “Not Found” error displayed on the page instead.

All files in your asset directory will be served online and accessible to anyone: do not place sensitive files in it.

Though one may create multiple such directory and correspondingly use `addResourcePath` to specify multiple paths and prefixes, one will routinely specify a single one, named “assets” or “static,” which contains multiple sub-directories, one for each type of static file to obtain a directory that looks something like the tree below. This is, however, an unwritten convention which is by no means forced upon the developer: do as you wish.

```
assets/
  js/
    script.js
  css/
    style.css
  img/
    pic.png
```

At this stage we have made the JavaScript file we created accessible by the clients but we still have to source this file in our ui as currently this file is, though served, not used by our application. Were one creating a static HTML page one would use the `script` to `src` the file in the `head` of the page.

```
<html>
  <head>
    <!-- source the JavaScript file -->
    <script src="path/to/script.js"></script>
  </head>
  <body>
    <p id="content">Trying JavaScript!</p>
  </body>
</html>
```

In shiny we write the ui in R and not in HTML (though this is also supported). Given the resemblance between the names of HTML tags and shiny UI functions it's pretty straightforward, the html page above would look something like the shiny ui below.

```
library(shiny)

ui <- fluidPage(
  tags$head(
    tags$script(src = "path/to/script.js")
  ),
  p(id = "content", "Trying JavaScript!")
)
```

Note that we use the `tags` object which comes from the shiny package and includes HTML tags that are not exported as standalone functions. For instance, you can create a `<div>` in shiny with the `div` function but `tags$div` will also work. This can now be applied to the shiny application, the `path/to/script.js` should be changed to `files/script.js` where `files` is the prefix we defined in `addResourcePath`.

```
# app.R
library(shiny)

# serve the files
addResourcePath(prefix = "files", directoryPath = "assets")

ui <- fluidPage(
  tags$head(
    tags$script(src = "files/script.js")
  ),
  h1("R and JavaScript")
)

server <- function(input, output){}

shinyApp(ui, server)
```

From the browser, inspecting page (right click > inspect > console tab) one should see “Hello JS!” in the console which means our application correctly ran the code in our JavaScript file.

Integration

The following sections will walk you through an example in which one first starts with a basic hello-world-like JavaScript example running in shiny and ends with a fully-fledged R package that brings new functionalities to shiny by integrating a third party library.

We first build an application that passes a message from the R server to the client to display said message as a vanilla JavaScript alert (pop-up), then send back to the R server whether the user has clicked “OK” on the alert. Finally we replicate that but with an external library and wrap the whole project into a fully functional package. This package will look similar to others that provide JavaScript integrations with shiny such as shinyjs (?) or waiter (?).

First, let’s write a barebone shiny application which includes the JavaScript code which opens an alert.

```
library(shiny)

ui <- fluidPage(
  tags$script(
    "alert('Hello from JavaScript');"
  ),
  h1("Hello")
)

server <- function(input, output, session){}

shinyApp(ui, server)
```

One thing important to note for later is that alerts will always block the execution of code which allows making sure some code is only run with user consent or the user being aware of the consequences.

```
alert('delete everything?');
deleteEverythingOnlyIfUserOK();
```

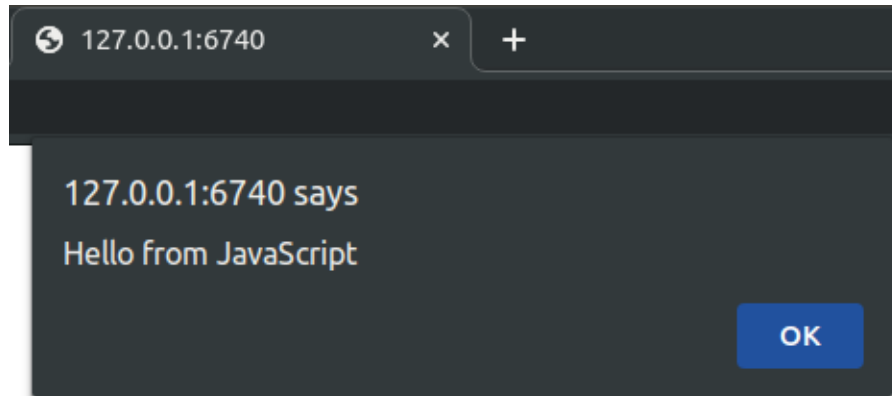


FIGURE 4.1: JavaScript alert in shiny

From R to JavaScript

Now that we have a simple alert displayed in the application we can tie it with our R server; the alert should display a message sent by the R server, this would enable, for instance, displaying a message taken from a database or a user input. As might be expected there are two functions required to do so, an R function and its JavaScript complementary: one to send the data from the server and another to catch said data from the client and display the alert.

Let us start by writing the R code to send the data, thankfully very little is required of the developer. One can send data from the R server to the client from the `session` object using the `sendCustomMessage` method. The method takes two arguments, first an identifier (where to send the data to), second the actual data to send to JavaScript.

```
server <- function(input, output, session){  
  # set the identifier to send-alert  
  session$sendCustomMessage(type = "send-alert", message = "Hi there!")  
}
```

This effectively sends our message to the JavaScript client but we are yet to use that message JavaScript-side so the application still displays the same alert on load. We can add a “handler” for the identifier we defined (`send-alert`) which will do something with the message we sent from the server. This is done with the `addCustomMessageHandler` method from the `Shiny` object where the first argument is the identifier and the second is the function that handles the

message, generally a function that takes a single argument: the data sent from the server.

```
tags$script(  
  "Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {  
    alert(message);  
  });"  
)
```

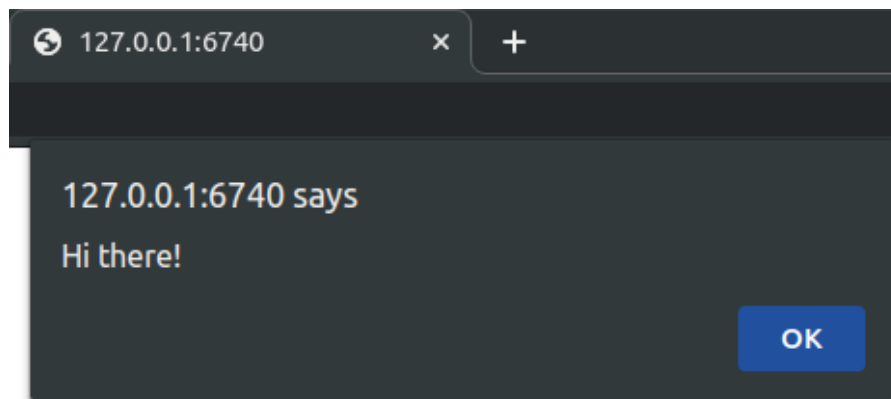


FIGURE 4.2: Alert sent from shiny server

This enables you to pass a message that is taken from a database for instance, or as shown below from a user input, to the alert.

```
library(shiny)  
  
ui <- fluidPage(  
  tags$script(  
    "Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {  
      alert(message);  
    });"  
  ),  
  h1("Hello"),  
  textInput("text", "Text to show in alert"),  
  actionButton("submit", "Show alert")  
)  
  
server <- function(input, output, session){  
  observeEvent(input$submit, {  
    session$sendCustomMessage(type = "send-alert", message = input$text)
```

```

    })
  }

shinyApp(ui, server)

```

In the application above, notice the path that our message follows: it goes from the client to the server which sends it back to the client. This might be considered suboptimal by some as it is not necessary to use the server as intermediary (in this example at least). Though there is some truth to this the above will work perfectly fine—and our aim here is to make JavaScript work with R—not alongside it.

From JavaScript to R

Imagine if you will that instead of displaying a somewhat anodyne alert it was one that actually mattered where the user is warned that clicking “OK” will execute an irreversible action like the deletion of a record. In order to implement this the server would need to “know” whether the user has clicked said “OK” button. To do so one needs to pass data from the client to the server.

This can be done by defining a new *simplified* shiny input. While one can define a fully-fledged shiny input that can be registered, updated, etc. there is also a simplified version of the latter which allows sending reactive input value to the server where it can be used just like any other inputs (`input$id`). The value of the input can be defined using the `setInputValue` method which takes the id of the input and the value to give it.

```

tags$script(
  "Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {

    // show alert
    alert(message);

    // set to true when clicked OK
    Shiny.setInputValue('delete_alert', true);
  });"
)

```

As mentioned earlier `alert` blocks code execution, therefore the input value

will not be defined before the button “OK” is pressed. The server can now access the `input$delete_alert` input which is by default `NULL` and set to `TRUE` when the user has pressed “OK,” as done in the application below which prints the input to the console when the button is clicked.

```
library(shiny)

ui <- fluidPage(
  tags$script(
    "Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {
      alert(message);
      Shiny.setInputValue('delete_alert', true);
    });"
  ),
  h1("Hello")
)

server <- function(input, output, session){
  session$sendCustomMessage(type = "send-alert", message = "Deleting a record!")

  observeEvent(input$delete_alert, {
    # print TRUE when button is clicked
    print(input$delete_alert)
  })
}

shinyApp(ui, server)
```

External Library

Thus far this chapter has covered both ways data travels between JavaScript and R in Shiny. However, the alerts displayed in the previous sections are rather hideous and, though demonstrates how both languages can work together within shiny, comes short of illustrating how to make use of external libraries, which is frequently performed as one progressively learn a language.

Let’s exploit an external library to improve upon the work done so far: `jBox`³ allows displaying modals (pop-ups), very similar to the vanilla JavaScript alerts, but much better looking and with additional functionalities.

³<https://github.com/StephanWagner/jBox>

The very first thing to do is to import jBox in our project, we could download the files and use them as described in the previous static files section but it comes with very convenient CDNs detailed in the get-started page of the documentation⁴.

```
<script src="https://code.jquery.com/jquery-3.5.1.min.js"></script>
<script src="https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.js"></script>
<link href="https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.css" rel="stylesheet">
```

Note that the “j” in jBox stands for jQuery which is already a dependency of shiny itself, there is therefore no need to import it, on the contrary one should not in order to avoid clashes. We can adapt the ui of the shiny application built up to this point to import the dependencies, keeping the handler, to obtain the code below.

```
ui <- fluidPage(
  tags$head(
    tags$script(
      src = "https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.js"
    ),
    tags$link(
      rel = "stylesheet",
      href = "https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.css"
    ),
    tags$script("Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {
      // TO DO: code jBox
    });")
  )
)
```

The jBox library comes with numerous features to display tooltips, modals, notices, and more, which would make for too long a chapter; only notices shall be covered here. Further down the get-started document⁵ lies an example of a jBox notice.

```
new jBox('Notice', {
  content: 'Hurray! A notice!',
  color: 'blue'
});
```

Let us copy that in the placeholder of the shiny ui already put together.

⁴https://stephanwagner.me/jBox/get_started

⁵https://stephanwagner.me/jBox/get_started#notices

```

library(shiny)

ui <- fluidPage(
  tags$head(
    tags$script(
      src = "https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.js"
    ),
    tags$link(
      rel = "stylesheet",
      href = "https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.css"
    ),
    tags$script("Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {
      new jBox('Notice', {
        content: 'Hurray! A notice!',
        color: 'blue'
      });
    });")
  )
)

server <- function(input, output, session){
  session$sendCustomMessage(type = "send-alert", message = "Deleting a record!")
}

shinyApp(ui, server)

```

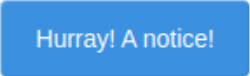


FIGURE 4.3: First jBox Notice

With some minor changes the application can display the message passed, one only needs to replace ‘Hurray! A notice!’ with the `message` variable.

```

tags$script("Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {
  new jBox('Notice', {

```

```
    content: message,  
    color: 'blue'  
  });  
});")
```

This though only allows passing a single variable, the message, to JavaScript but jBox has many more options.

Serialisation

Let's delve deeper into the communication between the server and the front-end to understand how we can further customise the notice displayed, e.g.: change the colour.

```
{  
  content: 'Hurray! A notice!',  
  color: 'blue'  
}
```

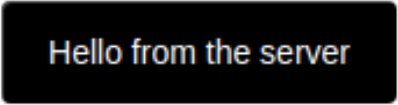
Notice that the jBox notice takes a JSON object containing the options that define said notice to display, including but not limited to the message. The most straightforward way to make all those options accessible to the server is to construct that list of options server-side before sending it to the front-end. For instance the JSON of options displayed above would look like the R list below.

```
list(  
  content = 'Hurray! A notice!',  
  color = 'blue'  
)
```

Therefore one could construct this list server-side and use it in jBox straight away, without any further processing from the client. Doing so means we can simplify the JavaScript to `new jBox('Notice', message);` rather than use `message` within the JSON.

```
library(shiny)
```

```
ui <- fluidPage(  
  tags$head(  
    tags$script(  
      src = "https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.js"  
    ),  
    tags$link(  
      rel = "stylesheet",  
      href = "https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.css"  
    ),  
    tags$script("Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {  
      // use notice send from the server  
      new jBox('Notice', message);  
    });")  
  )  
)  
  
server <- function(input, output, session){  
  
  # define notice options  
  notice = list(  
    content = 'Hello from the server',  
    color = 'black'  
  )  
  
  # send the notice  
  session$sendCustomMessage(type = "send-alert", message = notice)  
}  
  
shinyApp(ui, server)
```



Hello from the server

FIGURE 4.4: Customised jBox Notice

This sets a solid basis to further integrate other functionalities of jBox.

Events & Callbacks

In the example of the vanilla JavaScript alert one could simply place a line of code after the `alert()` function in order to “tell” the server whether the button on the alert had been clicked. This was feasible because `alert` stops the execution of code, this is, however, rather uncommon in JavaScript. What is far more used are events and callback functions which are triggered upon an action is performed by the user (like the click of a button) or when other interesting things happen in the code. jBox provides numerous such events⁶: callback functions can be used when a modal is closed or when it is created for instance.

The concept of the callback function is not totally foreign to R albeit rarely used. Shiny comes with such functions, e.g.: `shiny::onStop`. This allows having a function be triggered when the application exits (useful to close database connections for instance).

```
server <- function(input, output){
  shiny::onStop(
    # callback function fired when app is closed
    function(){
      cat("App has been closed")
    } for instance
  )
}
```

In jBox, these callback functions are included in the JSON of options, below the `onClose` event is fired when the notice is closed.

```
{
  content: 'Alert!',
  onClose: function(){
    // Fired when closed
    console.log('Alert is closed');
  }
}
```

⁶<https://stephanwagner.me/jBox/options#events>

This raises one issue, one cannot truly serialise an object of type function (there are exceptions later in the book).

```
# try to serialise an R function
jsonlite::toJSON(function(x){x + 1})
```

```
## ["function (x) ","{","    x + 1","}"]
```

One solution is to append the callback function to the object of options JavaScript-side.

```
tags$script("Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {
  // append callback
  message.onClose = function(){
    Shiny.setInputValue('alert_close', true);
  }
  new jBox('Notice', message);
});")
```

Placing a function inside a JSON object is common in JavaScript, in contrast with R where though it works is rarely if ever done (outside of reference class/R6). The above JavaScript code to append the callback function could look something like the snippet below in R.

```
message <- list(content = "hello")
message$onClose <- function(x){
  x + 1
}

message$onClose(2)
```

```
## [1] 3
```

That done it can be incorporated into the application built thus far. Something interesting could be done server-side but for the sake of this example we merely print the value of the input to the R console.

```
library(shiny)

ui <- fluidPage(
  tags$head(
    tags$script(
      src = "https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.js"
    ),
```

```

tags$link(
  rel = "stylesheet",
  href = "https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.css"
),
tags$script("Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {
  message.onClose = function(){
    Shiny.setInputValue('alert_close', true);
  }
  new jBox('Notice', message);
});")
)
)

server <- function(input, output, session){

  # define notice options
  notice = list(
    content = 'Hello from the server',
    color = 'black'
  )

  # send the notice
  session$sendCustomMessage(type = "send-alert", message = notice)

  # print the output of the alert_close event (when fired)
  observeEvent(input$alert_close, {
    print(input$alert_close)
  })
}

shinyApp(ui, server)

```

As a Package

If confused by any of the following please visit the package development section⁷ in the first chapter.

Packages are a fundamental part of R and allow conveniently sharing and

⁷[package-development](#)

reusing code. The work done so far is probably fitting for a single application but one should not have to reproduce all of that every time one wants to use jBox in a shiny application: we ought to wrap these functionalities into a handy package that can be used, reused and shared with others. Moreover, this will benefit from all the other advantages that R packages bring to code such as documentation, reproducibility, and tests.

Before we delve into building the package let us think through what it should include. The application using jBox gives some indication as to what the package will look like. Users of the package should be able to reproduce what is executed in the application, namely import dependencies (including the message handler) as well as send data to the JavaScript front-end. Finally, we shall provide, with the package, a static directory of dependencies to avoid relying on the CDNs as this ensures reproducibility (code hosted online by third-party might change and break the package). Concretely, the package will export a `useJbox` function to be placed in the shiny ui to import the dependencies (essentially replacing the `tags$*`) and a function to send the alert from the server to the client.

Let's start by creating an R package, here we name it "jbox," after the JavaScript of the same name, partly because the author lacks creativity: feel free to name it however you want.

```
usethis::create_package("jbox")
```

Dependencies

If confused by any of the following please visit the [static files section](#) of this chapter.

The very first thing that is required are the dependencies without which nothing can work, let's create a directory of static assets, download and place the jBox CSS and JavaScript files within it. We create the directory "inst" as per the R package convention and within it create another to hold our assets.

```
# create directories
dir.create("inst/assets", recursive = TRUE)
```

The jBox files can be downloaded from the CDN and placed within the directory that was created above. Moreover, we also create an empty JavaScript file that will eventually contain the custom JavaScript code that "connects" R to JavaScript.

```
# URLs of CDNs
js_dep <- "https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.js"
css_dep <- "https://cdn.jsdelivr.net/gh/StephanWagner/jBox@v1.2.0/dist/jBox.all.min.css"

# download
download.file(js_dep, destfile = "./inst/assets/jBox.all.min.js")
download.file(css_dep, destfile = "./inst/assets/jBox.all.min.css")

# create file to eventually hold custom JavaScript
file.create("./inst/assets/custom.js")
```

This done one should obtain a directory that looks similar to the tree below (some files and folders omitted for brevity).

```
DESCRIPTION
R/
inst/
  assets/
    jBox.all.min.js
    jBox.all.min.css
```

Next, one needs to have those files served, the user of the package could be asked to use `shiny::addResourcePath` but it's very inelegant, this should be built-in the package so the user does not even have to know this happening in the background. Therefore, we ought to ensure the static files are served when the user uses the package. Packages can optionally run functions when it is loaded or attached, Hadley Wickham writes about it extensively in the namespace chapter of his book on R packages⁸. By convention, these functions are placed in a `zzz.R` file.

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

The difference between loading and attaching a package can be subtle, in this case it's probably best to run the function when the package is loaded using `onLoad` which the R Packages book describes as:

Loading will load code, data and any DLLs; register S3 and S4 methods; and run the `.onLoad()` function. After loading, the package is available in memory, but because it's not in the search

⁸<http://r-pkgs.had.co.nz/namespace.html>

path, you won't be able to access its components without using `::`. Confusingly, `::` will also load a package automatically if it isn't already loaded. It's rare to load a package explicitly, but you can do so with `requireNamespace()` or `loadNamespace()`.

The `addResourcePath` function should thus be placed within the `.onLoad` function, this way the files are served by shiny when the package is loaded. Note the few changes below, we refer to the path using `system.file` (detailed in the [Package Development section](#)) and change the the prefix to `jbox-assets` to avoid the url serving our static files to clash with others.

```
# R/zzz.R
.onLoad <- function(libname, pkgname) {
  shiny::addResourcePath(
    "jbox-assets",
    system.file("assets", package = "jbox")
  )
}
```

This serves the file and allows not having to explicitly use `addResourcePath` but the package nonetheless needs to feature a function to let the user import them into their application.

```
#' Import Dependencies
#' @export
usejBox <- function(){
  shiny::tags$head(
    shiny::tags$script(src = "jbox-assets/jBox.all.min.js"),
    shiny::tags$link(rel = "stylesheet", href = "jbox-assets/jBox.all.min.css"),
    shiny::tags$script(src = "jbox-assets/custom.js")
  )
}
```

Users of the package can place the function defined above in the UI of their application to import the dependencies.

R Code

Not much changes from what was written before, however, it poses interesting questions with regard to the interface we want to provide users. From the

user's perspective the core of the package is the function that actually sends an alert to the clients, here created in `R/core.R`.

```
#' Create an Alert
#' @export
send_alert <- function(content = "alert", color = "blue", session){
  # define notice options
  notice = list(content = content, color = "black")

  # send the notice
  session$sendCustomMessage(type = "send-alert", message = notice)
}
```

One could leave it at the function above, it could be sufficient in providing a functional R package. However one could improve somewhat on it, currently, the function requires the `session` object, which confuses many, most R developers, including I, have little understanding of it. This can be mitigated by providing a default using `shiny::getDefaultReactiveDomain` which, notwithstanding its grandiose name, simply returns the shiny `session`. This means that function has to be run from a shiny server function but that is no limitation in this case.

```
#' Create an Alert
#' @export
send_alert <- function(content = "alert", color = "blue", session = shiny::getDefaultReactiveDomain){
  # define notice options
  notice = list(content = content, color = "black")

  # send the notice
  session$sendCustomMessage(type = "send-alert", message = notice)
}
```

This covers most of the R code that needs to be written, though we will come back to it shortly on as we uncover an interesting caveat.

JavaScript Code

Onto the JavaScript code, the `custom.js` to host said code is already created but remains empty. Simply using the code that was written previously will do the job for now.

```
// custom.js
Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {
  message.onClose = function(){
    Shiny.setInputValue('alert_close', true);
  }
  new jBox('Notice', message);
});
```

Shortcoming

At this stage one has a fully functional package: document, load the functions, and it can be used.

```
devtools::document()
devtools::load_all()

library(shiny)

ui <- fluidPage(
  usejBox(),
  verbatimTextOutput("callback")
)

server <- function(input, output){
  send_alert("Hello from the server!")

  output$callback <- renderPrint({
    paste("Is the alert closed: ", input$alert_close)
  })
}

shinyApp(ui, server)
```

However, while the above will work for a single alert it will run into issues when creating more than one alert as multiple alerts will set a value for a single input (`input$alert_close`). This is can be remedied to.

Input

The package needs to provide the user a way to distinguish between alerts in order to be able to observe the correct inputs server-side. A simple solution

consists in asking the user to provide an identifier (`id`). This identifier must be passed to the JavaScript client so the function can dynamically set the input value for that identifier, therefore it is included in the `message`, below we do so in such a way that the original JSON of options remains unchanged.

```
#' Create an Alert
#' @export
send_alert <- function(id, content = "alert", color = "blue", session = shiny::getDefaultReactiveDomain()) {
  # define notice options
  notice = list(content = content, color = "black")

  # add id
  message <- list(id = id, notice = notice)

  # send the notice
  session$sendCustomMessage(type = "send-alert", message = message)
}
```

Now one can adapt the JavaScript code to make use of the identifier. One needs to include said identifier in the name of the input the value of which is set, below we concatenate before the original input name. This is not forced upon the developer but is a convention, packages like DT and plotly approach the issue the same way: `id + name_of_input`. The event is thus now appended to `message.notice`, which is also used when creating the jBox alert.

```
// custom.js
Shiny.addCustomMessageHandler(type = 'send-alert', function(message) {
  message.notice.onClose = function(){
    console.log("close");
    Shiny.setInputValue(message.id + '_alert_close', true);
  }
  new jBox('Notice', message.notice);
});
```

Wrapping up

Building and installing the package will now provide the user an interface demonstrated below.

```
library(jbox)
library(shiny)
```

```
ui <- fluidPage(  
  usejBox(),  
  verbatimTextOutput("callback")  
)  
  
server <- function(input, output){  
  send_alert("myid", "Hello from the server!")  
  
  output$callback <- renderPrint({  
    paste("Is the alert closed: ", input$myid_alert_close)  
  })  
}  
  
shinyApp(ui, server)
```

It must be noted that though the package will be fully functional it will not pass any checks as documentation is poor and the DESCRIPTION incomplete. The API provided to the user is probably subpar in places, namely with the use of the id, which, unless the user needs to observe the respective input, is not necessary: forcing the user to provide it is not great design, consider making this optional.

Exercises

If one wants to create such packages or make extensive use of such integrations with Shiny, it is greatly encouraged to explore it further, namely by improving the API constructed and extending the functionalities (jBox comes with much more than just alerts), even integrate a JavaScript of your choice. At the time of writing this there is not package providing integration with jBox, if that is not of interest to the reader below are some other great libraries that are yet to be packaged in R and would greatly benefit the R community:

- micromodal.js⁹ - tiny, dependency-free javascript library for creating accessible modal dialogs
- hotkeys¹⁰ - Capturing keyboard inputs
- handtrack.js¹¹ - realtime hand detection

⁹<https://github.com/Ghosh/micromodal>

¹⁰<https://github.com/jaywcjlove/hotkeys>

¹¹<https://github.com/victordibia/handtrack.js>

- Rsup-progress¹² - simple progress bars

¹²<https://github.com/skt-t1-byungi/rsup-progress>



Part IV

Data Visualisation



5

HTML widgets

In this chapter we cover the integration of JavaScript with R using the `htmlwidgets` package, which focuses on libraries that produce a visual output, it is often used for data visualisation but is not limited to it.

As in previous chapters we mainly learn by example, building multiple widgets of increasing complexity as we progress through the chapter. Before writing the first widget, we explore JavaScript libraries that make great candidates for `htmlwidgets` and attempt to understand how they work to grasp what is expected from the developer in order to integrate them with R. Finally, we build up on the previous chapter to improve how HTML widgets work with shiny.

Candidate Libraries

Before going down the rabbit hole it is good to take a look at the types of libraries one will work with. As `htmlwidgets`' main client are JavaScript visualisation libraries let us take a look at some such popular libraries and briefly look at how they work and what they have in common. This will greatly help conceptualise what one is trying to achieve in this chapter.

Plotly

`Plotly.js`¹ is probably one of the more popular out there, it provides over 40 fully customisable chart types, many of which are very sophisticated. This is indeed the JavaScript library used by the R package of the same name: `plotly`.

Looking at the code presented in the “Get Started” guide reveals just how convenient the library is. One must import `plotly`, of course, then have a `<div>` where to visualisation will be placed, then, using `Plotly.newPlot`, create the

¹<https://plotly.com/javascript/>

actual visualisation by passing it first the element previously mentioned and a JSON of options.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="" xml:lang="">

<head>
  <!-- Import library -->
  <script src="plotly-latest.min.js"></script>
</head>

<body>
  <!-- div to hold visualisation -->
  <div id="chart" style="width:600px;height:250px;"></div>

  <!-- Script to create visualisation -->
  <script>
    el = document.getElementById('chart');
    Plotly.newPlot(el, [{
      x: [1, 2, 3, 4, 5],
      y: [1, 2, 4, 8, 16] }]
    );
  </script>
</body>

</html>
```

Now let's look at how another popular library does it.

Highchart.js

Highcharts² is another library which allows creating gorgeous visualisation, maps, and more, it's also very popular albeit not being entirely open-source.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="" xml:lang="">

<head>
  <!-- Import library -->
  <script src="highcharts.js"></script>
</head>
```

²<https://www.highcharts.com/>

```
<body>
  <!-- div to hold visualisation -->
  <div id="chart" style="width:100%; height:400px;"></div>

  <!-- Script to create visualisation -->
  <script>
    var myChart = Highcharts.chart('chart', {
      xAxis: {
        categories: ['Apples', 'Bananas', 'Oranges']
      },
      series: [{
        name: 'Jane',
        data: [1, 0, 4]
      }, {
        name: 'John',
        data: [5, 7, 3]
      }]
    });
  </script>
</body>

</html>
```

The above is very similar to what `plotly.js` requires: import libraries, create a `<div>` where to put the visualisation, and, to create the chart, run a function which also takes the id of the div where to place the chart and a JSON of options defining the actual chart, including the data.

Chart.js

`Chart.js`³ is yet another library which to draw standard charts popular for its permissive license and convenient API.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="" xml:lang="">

<head>
  <!-- Import library -->
  <script src="Chart.min.js"></script>
</head>
```

³<https://www.chartjs.org/>

```

<body>
  <!-- canvas to hold visualisation -->
  <canvas id="chart" width="400" height="400"></canvas>

  <!-- Script to create visualisation -->
  <script>
    var el = document.getElementById('chart').getContext('2d');
    var myChart = new Chart(el, {
      type: 'bar',
      data: {
        labels: ['Red', 'Blue', 'Yellow', 'Green', 'Purple', 'Orange'],
        datasets: [{
          label: '# of Votes',
          data: [12, 19, 3, 5, 2, 3]
        }]
      }
    });
  </script>
</body>

</html>

```

We again observe a very similar structure as with previous libraries. The library is imported, instead of a `div` `chart.js` uses a `canvas`, and the visualisation is also created from a single function which takes the canvas as first argument and a JSON of options as second.

Hopefully this reveals the repeating structure such libraries tend to follow and also hints at should be reproduced, to some extent at least, using R.

How it works

Imagine there is no such package as HTML widgets to help create interactive visualisations from R: how would one attempt to go about it?

An interactive visualisation using JavaScript will be contained within an HTML document, therefore it would probably have to be created first. Secondly, the visualisation that is yet to be created likely relies on external libraries, these would need to be imported in the document. The document should also include an HTML element (e.g.: `<div>`) to host said visualisation. Then data would have to be serialised in R and embedded into the document

where it should be read by JavaScript code that uses it to create the visualisation. Finally all should be managed to work seamlessly across R markdown, shiny, and other settings.

Thankfully the `htmlwidgets` package is there to handle most of this. Nonetheless, it is important to understand that these operations are undertaken (to some degree) by `htmlwidgets` as it greatly helps use the package.

Must remember when building HTML widgets:

- Import dependencies
- Create an html element to hold visualisation
- Serialise R data to JSON
- Handle JSON data to produce visualisation

The Scaffold

With some vague understanding of how such widgets work internally one is ready to “scaffold” one with the aim of rummaging through its components to grasp a greater understanding of how such interactive outputs are actually produced. The way one sets up such a package is stunningly simple. Below we create a package named “playground” which will be used to mess around and explore. Though one could probably create widgets outside of an R package, it would only make things more complicated.

```
create_package("playground")
```

Then, from the root of the package created, we scaffold a widget which we call “play”.

```
htmlwidgets::scaffoldWidget("play")
```

This function puts together the minimalistic structure necessary to implement an HTML widget and opens `play.R`, `play.js` and `play.yaml` in the RStudio IDE or the default text editor. These files are named after the widget and will form the core of the package. The R file contains core functions of the R API, namely the `play` function which creates the widget itself, and the `render*` and `*output` functions that handle the widget in the shiny server and UI respectively. The `.js` file contains JavaScript functions that actually generate the visual output.

```
devtools::document()
devtools::load_all()
```

It might be hard to believe, but at this stage one already has a fully functioning widget ready to use after documenting, and building the package. Indeed, the `play.R` file that that was created contains a function named “play” which takes, amongst other arguments, a message.

```
play(message = "This is a widget!")
```

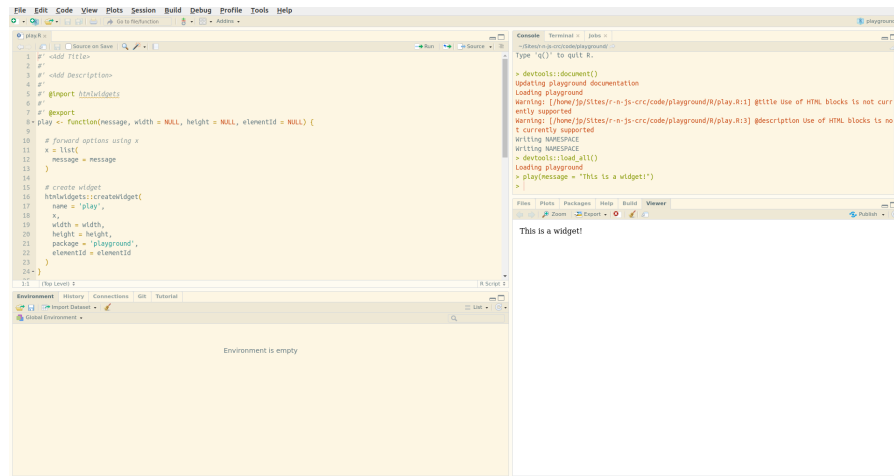


FIGURE 5.1: First HTML widget

This displays the message in the RStudio “Viewer,” or the your default browser which indicates that the function does indeed create an HTML output. One



can use the the button located in the top right of the RStudio “Viewer” to open the message in web browser which can prove very useful to look under the hood of the widgets for debugging.

The Output

With an out-of-the-box HTML widget package one can start exploring the internals to understand how it works. Let's start by retracing the path taken by the message written in R to its seemingly magical appearance in HTML. The `play` function previously used, takes the `message` wraps it into a list which is then used in `htmlwidgets::createWidget`.

```
# forward options using x
x = list(
  message = message
)
```

Wrapping a string in a list might seem unnecessary but one will eventually add variables when building a more complex widget, starting with a list makes it easier to add them later on.

To investigate the widget we should look under the hood; the source code of the created (and rendered) output can be accessed in different ways, 1) by right-clicking on the message displayed in the RStudio Viewer and selecting “Inspect element,” or 2) by opening the visualisation in your browser using the



button located in the top right of the “Viewer,” and in the browser right clicking on the message to select “Inspect.” The latter is advised as web browsers such as Chrome or Firefox provide much friendlier interfaces for such functionalities as well as shortcuts to inspect or view the source code of a page.

Below is a part of the `<body>` of the output of `play("This is a widget!")` obtained with the method described in the previous paragraph.

```
<div id="htmlwidget_container">
  <div id="htmlwidget-c21cca0e76e520b46fc7" style="width:960px;height:500px;" class="play">
</div>
<script type="application/json" data-for="htmlwidget-c21cca0e76e520b46fc7">{"x":{"message"
```

One thing the source code of the rendered output reveals is the element (`div`) created by the `htmlwidgets` package to hold the message (the class name is identical to that of the widget, `play`), as well as, below it, in the `<script>` tag, the JSON object which includes the `x` variable used in the `play` function. The `div` created bears a randomly generated `id` which one can define when creating the widget using the `elementId` argument.

```
# specify the id
play("This is another widget", elementId = "myViz")
```

```
<!-- div bears id specified in R -->
<div id="myViz" style="width:960px;height:500px;" class="play html-widget">This is another
```

You will also notice that this affects the `script` tag below it, the `data-for` attribute of which is also set to “myViz,” this indicates that it is used to tie the JSON data to a `div`, essential for `htmlwidgets` to manage multiple visualisation in R markdown or Shiny for instance. Then again, this happens in the background without the developer (you) having to worry about it.

```
<script type="application/json" data-for="myViz">{"x":{"message":"This is a widget!"},"eval":{}}
```

Inspecting the output also shows the dependencies imported, these are placed within the `head` HTML tags at the top of the page.

```
<script src="lib/htmlwidgets-1.5.1/htmlwidgets.js"></script>
<script src="lib/play-binding-0.0.0.9000/play.js"></script>
```

This effectively imports the `htmlwidgets.js` library as well as the `play.js` file, and were the visualisation depending on external libraries they would appear alongside those. Peeking inside the `play.js` file located at `inst/htmlwidgets/play.js` reveals the code below we see:

```
// play.js
HTMLWidgets.widget({

  name: 'play',

  type: 'output',

  factory: function(el, width, height) {

    // TODO: define shared variables for this instance

    return {

      renderValue: function(x) {

        // TODO: code to render the widget, e.g.
```

```

        el.innerText = x.message;

    },

    resize: function(width, height) {

        // TODO: code to re-render the widget with a new size

    }

    };
}
});

```

However convoluted this may appear at first do not let that intimidate you. The **factory** function returns two functions, one of which, **resize**, is currently empty, let's therefore look at the other one first, **renderValue**: the function that in fact renders the visualisation. It takes an object **x** from which it accesses the “message” variable that it uses as text for object **el** (**el.innerText**). The object **x** passed to this function is actually the list of the same name that was built in the R function **play!** While in R one would access the **message** in list **x** with **x\$message** in JavaScript to access the **message** in the JSON **x** one writes **x.message**, only changing the dollar sign to a dot. Let's show this perhaps more clearly by printing the content of **x**.

```

console.log(x);
el.innerText = x.message;

```

We place **console.log** to print the content of **x** in the console, reload the package with **devtools::load_all** and use the function **play** again then explore the console from the browser (inspect and go to the “console” tab).

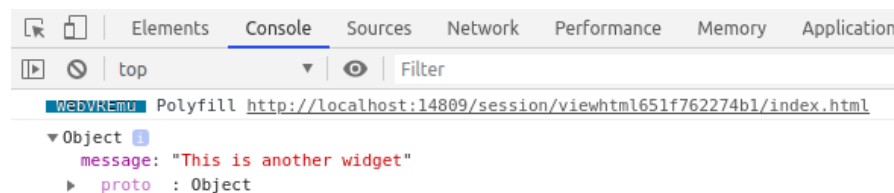


FIGURE 5.2: Console tab output

This displays the JSON object containing the message: it looks eerily similar to the list that was created in R (**x = list(message = "This is a widget!")**). What one should take away from this is that data that needs

to be communicated from R to the JavaScript function should be placed in the R list `x`. This list is serialised to JSON and placed in the HTML output in a `script` tag with a `data-for` attribute. This attribute indicates which widget the data is destined for. This effectively enables `htmlwidgets` to match the serialised data with the output elements: data in `<script data-for='viz'>` is to be used to create a visualisation in `<div id='viz'>`.

Before we move on to other things one should also grasp a better understanding of the `el` object, which can also be logged in the console.

```
console.log(x);
console.log(el);
el.innerText = x.message;
```



FIGURE 5.3: Console tab output

This displays the HTML element created by `htmlwidgets` that is meant to hold the visualisation, or in this case, the message. If you are familiar with JavaScript, this is the element that would be returned by `document.getElementById`. This object allows manipulating the element in pretty much any way imaginable, change its position, its colour, its size, or, as done here, to insert some text in its place. What's more one can access attributes of the object just like a JSON array. Therefore one can log the `id` of the element.

```
// print the id of the element
console.log(el.id);
el.innerText = x.message;
```

Making the modifications above and reloading the package, one can create a widget given a specific id and see it displayed in the console, e.g.: `play("hello", elementId = "see-you-in-the-console")`.

In an attempt to become more at ease with this setup let us change something and play with the widget. Out-of-the-box `htmlwidgets` uses `innerText`, which does very much what it says on the tin, it places text inside an element. JavaScript comes with another function akin to `innerText`, `innerHTML`. While the former only allows inserting text the former lets one insert any HTML.

```
el.innerHTML = x.message;
```

After changing the `play.js` file as above, and re-loading the package, one can use arbitrary HTML as messages.

```
play("<h1>Using HTML!</h1>")
```

Using HTML!

FIGURE 5.4: Widget output

That makes for a great improvement which opens the door to many possibilities. However, the interface this provides is unintuitive. Albeit similar, R users are more familiar with shiny and htmltools (?) tags than HTML tags, e.g.: `<h1></h1>` translates to `h1()` in R. The package should allow users to use those instead of forcing them to collapse HTML content in a string. Fortunately, there is a very easy way to obtain the HTML from those functions: convert it to a character string.

```
html <- shiny::h1("HTML tag")
```

```
class(html)
```

```
## [1] "shiny.tag"
```

```
# returns string
```

```
as.character(html)
```

```
## [1] "<h1>HTML tag</h1>"
```

Implementing this in the `play` function will look like this.

```
# forward options using x
```

```
x = list(
  message = as.character(message)
)
```

Reloading the package with `devtools::load_all` lets one use shiny tags as the message.

```
play(shiny::h2("Chocolate is a colour", style = "color:chocolate;"))
```

Chocolate is a colour

FIGURE 5.5: Using shiny tags

This hopefully provides some understanding of how `htmlwidgets` work internally and thereby helps building such packages. To recapitulate, an HTML document is created in which `div` is placed and given a certain id, this id is also used in a script tag that contains JSON data passed from R so that a JavaScript function we define can read that data in and use it to generate a visual output in a `div`. However, as much as this section covered, the topic of JavaScript dependencies was not touched, this is approached in the following section where we build another, more interesting widget, which uses an external dependency.

Typed.js

In this section we build a package called `typed`, which wraps the JavaScript library of the same name, `typed.js`⁴ that mimics text being typed. This builds upon many things we explored in the `playground` package.

```
usethis::create_package("typed")
htmlwidgets::scaffoldWidget("typed")
```

As done with candidate libraries, let's take a look at documentation of `typed.js`⁵ to see how `typed.js` works.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="" xml:lang="">
```

⁴<https://github.com/mattboldt/typed.js/>

⁵<https://github.com/mattboldt/typed.js/>

```
<head>
  <!-- Import library -->
  <script src="typed.js"></script>
</head>

<body>
  <!-- div to hold visualisation -->
  <div class="element"></div>

  <!-- Script to create visualisation -->
  <script>
    var typed = new Typed('.element', {
      strings: ['First sentence.', 'And a second sentence.']
    });
  </script>
</body>

</html>
```

The code above is not very different from what was observed in other libraries: the library is imported, there is a `<div>` where the output will be generated, and a script which also takes a selector and a JSON of options.

Dependency

Once the package created and the widget scaffold laid down we need to add the JavaScript dependency without which nothing can move forward. The documentation in the README of `typed.js`⁶ states that it can be imported like so.

```
<script src="https://cdn.jsdelivr.net/npm/typed.js@2.0.11"></script>
```

First, we will download the dependency, which consists of a single JavaScript file, instead of using the CDN as this ultimately makes the package more robust (more easily reproducible outputs and no requirement for internet connection). Below we place the dependency in a “typed” directory within the “htmlwidgets” folder.

```
dir.create("../inst/htmlwidgets/typed")
```

⁶<https://github.com/mattboldt/typed.js>

```
cdn <- "https://cdn.jsdelivr.net/npm/typed.js@2.0.11"
download.file(cdn, "../inst/htmlwidgets/typed/typed.min.js")
```

This produces a directory that looks like this:

```
.
DESCRIPTION
NAMESPACE
R
  typed.R
inst
  htmlwidgets
    typed
      typed.min.js
      typed.js
      typed.yaml
```

In `htmlwidgets` packages dependencies are specified in the `.yaml` file located at `inst/htmlwidgets` which at first contains a commented template.

```
# (uncomment to add a dependency)
# dependencies:
#   - name:
#     version:
#     src:
#     script:
#     stylesheet:
```

Let's uncomment those lines as instructed at the top of the file and fill it in.

```
dependencies:
  - name: typed.js
    version: 2.0.11
    src: htmlwidgets/typed
    script: typed.min.js
```

We remove the `stylesheet` entry as this package does not require any CSS files. The `src` specifies the path to the directory containing the scripts and stylesheets. This is akin to using the `system.file` function to return the full path to a file or directory within the package.


```
devtools::load_all()
```



```
system.file("htmlwidgets/typed", package = "typed")
#> "/home/me/packages/typed/inst/htmlwidgets/typed"
```

We should verify that this is correct by using the one R function the package features and check the source code of the output to verify that the `typed.js` is indeed imported. We thus run `typed("test")`, open the output in the browser



() and look at the source code of the page (right click and select “View page source”). At the top of the page one should see `typed.min.js` imported, click the link to ensure it correctly points to the dependency.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<style>body{background-color:white;}</style>
<script src="lib/htmlwidgets-1.5.1/htmlwidgets.js"></script>
<script src="lib/typed.js-2.0.11/typed.min.js"></script>
<script src="lib/typed-binding-0.0.0.9000/typed.js"></script>
...
```

JavaScript

On its official website⁷, `typed.js` gives the following example. The JavaScript function `Typed` takes two arguments, first the selector, the element to hold the output, second a JSON of options to specify what is being typed and a myriad of other things.

```
var typed = new Typed('.element', {
  strings: ["First sentence.", "Second sentence."],
  typeSpeed: 30
});
```

Let’s place it in the package by replacing the content of the `renderValue` in `typed.js` with the above.

```
...
renderValue: function(x) {
```

⁷<https://mattboldt.com/demos/typed-js/>

```

var typed = new Typed('.element', {
  strings: ["First sentence.", "Second sentence."],
  typeSpeed: 30
});

}
...

```

One could be tempted to run `devtools::load_all` but this will not work, namely because the function uses a selector that will not return any object; it needs to be applied to the div created by the widget not `.element`. As hinted at in the playground, the selector of the element created is accessible from the `el` object. As a matter of fact, we did log in the browser console the id of the created div taken from `el.id`. Therefore concatenating the pound sign and the element id produces the select to said element. (`.class, #id`)

```

// typed.js
...
renderValue: function(x) {

  var typed = new Typed('#' + el.id, {
    strings: ["First sentence.", "Second sentence."],
    typeSpeed: 30
  });

}
...

```

This should now work, run `devtools::load_all` followed by `typed("whatever")` and the JavaScript animated text will appear! It's not of any use just yet as the options, included the text being typed is predefined: the package is currently not making any use of the inputs passed from R. Below change the default strings to `x.message`.

```

// typed.js
...
renderValue: function(x) {

  var typed = new Typed('#' + el.id, {
    strings: x.message,
    typeSpeed: 30
  });

}

```

```
}
...
```

This, however, will cause issues as the `strings` options expects an array (vector) and not a single string. This is something often forgotten when working with R, there is no scalar values, in R a scalar is vector of length 1.

```
typed("does not work") # length = 1
typed(c("This", "will", "work")) # length > 1
```

One solution is to force the input into a list.

```
# typed.R
x = list(
  message = as.list(message)
)
```

At this juncture the package works but there is a salient issue with the way it handles options. Why build a list in R to reconstruct it in JavaScript manually. Since the options are serialised in R to JSON and that typed.js expects a JSON of options it is actually cleaner and more convenient to construct an R list that mirrors the JSON array so one can use it as-is in JavaScript.

In fact, renaming the `message` to `strings` effectively does this.

```
# typed.R
x = list(
  strings = as.list(message)
)
```

This allows greatly simplifying the code JavaScript side, making it much easier to add other options down the line, maintain, debug, and read.

```
// typed.js
...
renderValue: function(x) {
  var typed = new Typed('#' + el.id, x);
}
...
```

One can now add more options from the R code without having to alter any of the JavaScript. Let us demonstrate with the `loop` option.

```
typed <- function(message, loop = FALSE, width = NULL, height = NULL, elementId = NULL) {  
  
  # forward options using x  
  x = list(  
    loop = loop,  
    strings = as.list(message)  
  )  
  
  # create widget  
  htmlwidgets::createWidget(  
    name = 'typed',  
    x,  
    width = width,  
    height = height,  
    package = 'typed',  
    elementId = elementId  
  )  
}
```

HTML Element

As pointed out multiple times, the widget is generated in a `<div>`, which is works fine for most visualisation libraries. But we saw that `chart.js` requires placing it in a `<canvas>`, so one needs the ability to change that. It could be interesting to apply this to `typed.js` too as within a `<div>` it cannot be placed inline, using a ``, however, this would work.

This can be changed by placing a function named `nameOfWidget_html` which looked up by `htmlwidgets` and used if found. This function takes the three-dot construct `...` and uses them in an `htmltools` tag. The three-dots are necessary because internally `htmlwidgets` needs be able to pass arguments, such as the all too critical `id`.

```
typed_html <- function(...){  
  htmltools::tags$span(...)  
}
```