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

Photo by davisco on Unsplash

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 demonstrate to the reader the many great benefits one can reap by inviting JavaScript into their data science workflow. In that respect the book is not teaching one JavaScript but rather demonstrates how little JavaScript can greatly support and enhance R code. Therefore focus is on integrating external JavaScript libraries and only limited knowledge of JavaScript is required in order to learn from the book. Moreover, the book focuses on generalisable learnings so the reader can transfer takeaways from the book to solve real-world problems.

Throughout the book several shiny applications and R packages are put together as examples, all of these, along with the code for the entire book can be found on the Github repository: github.com/JohnCoene/javascript-for-r[1].

Premise

The R programming language has been propelled into web browsers with the introduction of packages such as shiny[2] (**?**) and rmarkdown[3] (**?**) which have greatly improved how R users can communicate complex insights by building interactive web applications and interactive documents. Yet most R developers are not familiar with one of web browsers' core technology: JavaScript. This books aims to remedy to that.

The book is nonetheless not about learning JavaScript, even though it is aimed at R developers with little knowledge of the language. This statement might

---

[1] https://github.com/JohnCoene/javascript-for-r
[2] https://shiny.rstudio.com/
[3] https://rmarkdown.rstudio.com/

appear counter-intuitive at first but, we hope, the book gradually reveals just how little JavaScript is actually needed in order to observe immense benefits otherwise locked away from R developers. This can be achieved by using Javascript only to do what R cannot, and rely on R, a language familiar to us, to do the heavy lifting. This, it must be said, might, in a few places, lead to code that will appear inefficient to trained JavaScript developer as instead of doing everything in the browser with JavaScript data might be passed from it to the server where R can be used but rest assured that the impact this has on performances is largely unnoticeable.

Importantly, the focus of the book truly is the integration of JavaScript with R where both languages either actively interact with one another, or where JavaScript enables doing things otherwise not accessible to R users—it is not merely about including JavaScript code that works alongside R.

Book Structure

1. The book opens with an introduction to better illustrate its premise, it provides rationales for using JavaScript in conjunction with R which it supports with existing R packages that make use of JavaScript and are available on CRAN. Then it briefly describes concepts essential to understand the rest of the book to ensure the reader can follow along. Finally, this part closes by listing the various methods with which one might make JavaScript work with R.

2. We explore existing integrations of JavaScript and R namely by exploring packages to grasp how these tend to work and the interface to JavaScript they provide.

3. A sizeable part of the book concerns data visualisation, it plunges into creating interactive outputs with the htmlwidgets package. This opens with a brief overview of how it works and libraries that make great candidates to integrate with the htmlwidgets package. Then a first, admittedly unimpressive, widget is built to look under the hood and observe the inner-workings of such outputs in order to grasp a better understanding of how htmlwidgets work. Next, we tackle a more substantial library that allows drawing arcs between countries on a 3D globe. Then, to demonstrate that such packages are not limited to visualisation and more broadly applies to interactive outputs, a second package is built, one that allows to mimic text being typed on the screen. The last two chapter go into more advanced topics such as security and resizing.

4. The fourth part of the book details how JavaScript can work with

Shiny. Once the basics out of the way, the second chapter builds the first utility to programmatically display notifications. Then we create a functionality to notify the user when the shiny server is busy running computations. This part of the book ends with an example of custom shiny output for a library that applies filters to images. Finally, shiny and htmlwidgets are (literally) connected by including additional functionalities in interactive visualisations when used with the shiny framework.

5. Finally, we look at how one can use some of the more modern JavaScript technologies such as Vue, React, and webpack with R—these can make the use of JavaScript more agile and robust.

6. Then the book delves into using JavaScript for computations, namely via the V8 engine and node.js. After a short introduction, chapters will walk the reader through various examples: a fuzzy search, a time format converter, and some basic natural language operations.

7. Next the book closes with examples of all the integrations explored previously. This involves recreating (a part of) the plotly package, building an image classifier, adding progress bars to a shiny application, building an app with HTTP cookies and running basic machine learning operations in JavaScript.

8. Finally, the book concludes with some noteworthy remarks on where to go next.

Acknowledgement

Disclaimer

This book is currently a work in progress.

Part I

Basics & Roadmap

1

Introduction

This book starts with a rationale for integrating JavaScript with R and supports it with examples, namely packages that use JavaScript and are available on CRAN. Then, we list the various ways in which one might go about making both languages work together. In the next chapter, we go over prerequisites and a review of concepts fundamental to fully understand the more advanced topics residing in the forthcoming chapters.

1.1 Rationale

Why blend two languages seemingly so far removed from each other? Well, precisely because they are 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.

Nevertheless, 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 by no means required to make those computations, they could be rewritten solely in R but that would be vastly more laborious than wrapping the JavaScript API in R as done by the package lawn.

```
library(lawn)

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

```
## <FeatureCollection>
##    Bounding box: -112.1 46.6 -112.0 46.6
```

---

[1] https://github.com/ropensci/lawn
[2] http://turfjs.org/

```
##    No. features: 2
##    No. points: 20
##    Properties:
##      values count
## 1 200, 600       2
## 2               0
```

Another great reason is that JavaScript can do things that R cannot, e.g.: run in the browser. Therefore one cannot natively create interactive visualisations with R. Plotly[3] (**?**) by Carson Sievert packages the plotly JavaScript library[4] to let one create interactive visualisations solely from R code.

```r
library(plotly)

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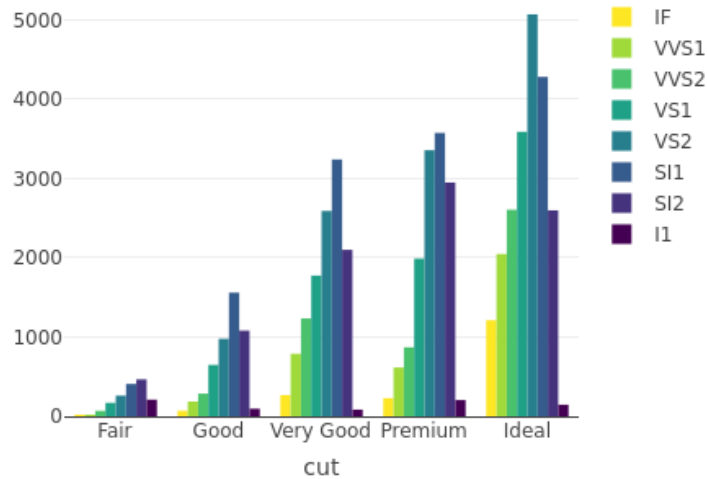

Finally, JavaScript can work together with R to improve how we communicate insights. One of the many ways in which Shiny stands out is that it lets one create web applications solely from R code with no knowledge of HTML, CSS, or JavaScript but that does not mean they can't extend Shiny, quite the

---

[3]https://plotly-r.com/
[4]https://plot.ly/

contrary. The waiter package[5] (**?**) integrates a variety of JavaScript libraries to display loading screens in Shiny applications.

```r
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)
```

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

1.2 Methods

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

---

[5]http://waiter.john-coene.com/
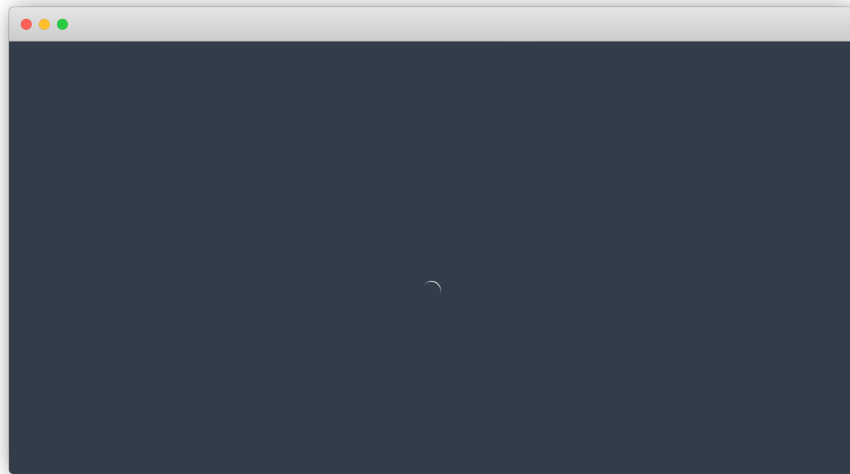
FIGURE 1.1: Waiter Output

1.2.1   V8

V8[6] 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 interfaces with the turf.js library.

```
library(V8)
```

```
## Using V8 engine 6.8.275.32-node.55
```

```
ctx <- v8()
```

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

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

---

[6]https://github.com/jeroen/v8

1.2.2   htmlwidgets

htmlwidgets[7] (**?**) specialises in wrapping JavaScript libraries that generate visual outputs. This is what packages such as plotly, DT[8] (**?**), highcharter[9] (**?**), 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.

1.2.3   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. This is how the package waiter mentioned previously internally works with R.

1.2.4   bubble

bubble[10] (**?**) by Colin Fay is a more recent R package, still under heady development but very promising: it lets one run node.js[11] code in R, comes with an interactive node REPL, the ability to install npm packages, and even an R markdown engine. It's similar to V8 in many ways.

```r
library(bubble)

n <- NodeSession$new()

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

```
## 4
```

---

[7]http://www.htmlwidgets.org/
[8]https://rstudio.github.io/DT/
[9]http://jkunst.com/highcharter/
[10]https://github.com/ColinFay/bubble
[11]https://nodejs.org/en/

1.3 Methods Amiss

Note that there are also two other prominent ways one can use JavaScript with R that are not covered in this book. The main reason being that they require great knowledge of specific JavaScript libraries, d3.js and React, and while these are themselves advanced uses of JavaScript, their integration with R via the packages listed below is rather straightforward.

1.3.1 reactR

ReactR[12] (?) is an R package that emulates very well htmlwidgets but specifically for the React framework[13]. Unlike htmlwidgets it is not limited to visual outputs and also provides functions to build inputs, e.g.: a drop-down menu (like `shiny::selectInput`). The reactable package[14] (?) uses reactR to enable building interactive tables solely from R code.

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

Sepal.Leng	Sepal.Widt	Petal.Leng	Petal.Wid
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

---

[12]https://react-r.github.io/reactR/
[13]https://reactjs.org/
[14]https://glin.github.io/reactable/

1.3.2   r2d3

r2d3[15] (**?**) by RStudio is an R package designed specifically to work with d3.js[16]. It is similar to htmlwidgets but works rather differently.

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


---

[15]https://rstudio.github.io/r2d3/
[16]https://d3js.org/
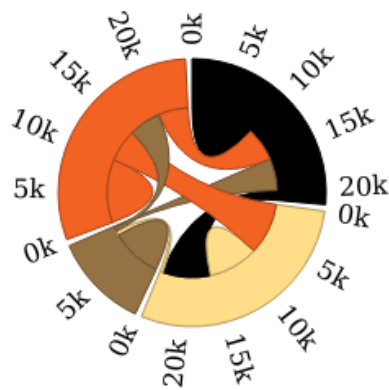
2

Prerequisites

The code contained in the following pages is approachable to readers with basic knowledge of R, but familiarity with package development using devtools[1] (**?**), the Shiny[2] framework (**?**), the JSON data format, and JavaScript are essential.

The reason for the former is that some of the ways one builds integrations with JavaScript naturally take the form of R packages. Also, R packages make sharing code, datasets, and anything else R-related extremely convenient, they come with a relatively strict structure, the ability to run unit tests, and much more. These have thus become a core feature of the R ecosystem and therefore are also used extensively in the book as we create several packages. The following section thus runs over the essentials of building a package to ensure everyone can keep up.

Then we briefly go through the JSON data format as it will be used to a great extent to communicate between R and JavaScript. Since both Shiny and JavaScript run in the browser they make for axiomatic companions; we'll therefore use Shiny extensively. Finally, there is an obligatory short introduction to JavaScript.

It is highly recommended to use the freely available RStudio IDE[3] to follow along as it makes a lot of things easier down the line.

2.1   R Package Development

Developing packages used to be notoriously difficult but things have greatly changed in recent years, namely thanks to the devtools (**?**), roxygen2 (**?**) and more recent usethis[4] (**?**) packages. Devtools is short for "developer tools," it is specifically designed to help creating packages; setting up tests, running checks, building and installing packages, etc. The second provides an all too convenient way to generate the documentation of packages, and usethis, more

---

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

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 Packages[5] solely dedicated to the topic.
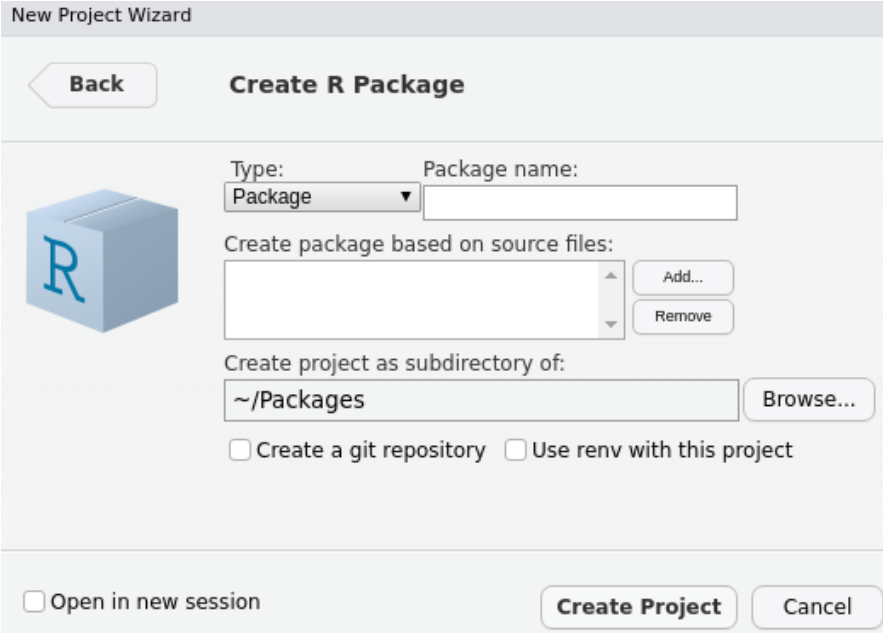
Start by installing those packages from CRAN, the roxygen2 package does not need to be explicitly installed as it is a dependency of devtools.

```r
install.packages(c("devtools", "usethis"))
```

2.1.1 Creating a Package

There are multiple ways to create a package. One could manually create every file, use the RStudio IDE, or create it from the R console with the usethis (**?**) package.

From the RStudio IDE go to `File > New Project > New Directory > R Package` then select "R package" and fill in the small form, namely name the package and specify the directory where it should be created.

FIGURE 2.1: Package creation wizard

But it could be argued that it's actually easier from the R console with the usethis package. The `create_package` function takes as first argument the

---

path to create the package. If you run it from RStudio a new project window
should open.

```
# creates a package named "test" in root of directory.
usethis::create_package("test")
```

```
  Creating 'test/'
  Setting active project to '/Packages/test'
  Creating 'R/'
  Writing 'DESCRIPTION'
Package: test
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
    * First Last <first.last@example.com> [aut, cre] (YOUR-ORCID-ID)
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
    pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1.9000
  Writing 'NAMESPACE'
  Changing working directory to 'test/'
  Setting active project to '<no active project>'
```

2.1.2 Metadata

Every R package includes a `DESCRIPTION` file which includes metadata
about the package. This includes a range of things like the license defin-
ing who can use the package, the name of the package, its dependen-
cies, and more. Below is the default created by the usethis package with
`usethis::create_package("test")`.

```
Package: test
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R:
    person(given = "First",
           family = "Last",
           role = c("aut", "cre"),
           email = "first.last@example.com",
           comment = c(ORCID = "YOUR-ORCID-ID"))
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
```

```
    pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1.9000
```

Much of this is outside the scope of the book. However, it is good to grasp how dependencies are specified. As packages are generally intended for sharing with others it is important to ensure users of the package meet the dependencies otherwise the package may not work in places. For instance, were we to create a package that relies on one or more functions from the stringr (**?**) package we would need to ensure people who install the package actually have it installed on their machine or those functions will not work.

```
# R/string.R
string_length <- function(string) {
  stringr::str_length(string)
}
```

Note that the function is preceded by its namespace with `::` (more on this later).

The `DESCRIPTION` file does this, it will make sure that the dependencies of the package are met by users who install it. We can specify such dependencies under `Imports` where we can list packages required separated by a comma.

```
Imports:
  stringr,
  dplyr
```

Then again, the usethis package also allows doing so consistently from the R console, which is great to avoid mishandling the `DESCRIPTION` file.

```
# add stringr under Imports
usethis::use_package('stringr')
```

One can also specify another type of dependencies under `Suggests`, other packages that enhance that enhance the package but are not required to run it. These, unlike package under 'Imports' are not automatically installed if missing which can greatly reduce overhead.

2.1.3   R code

An R package must follow a strict structure. R code must be placed in an `R/` directory so one should only find `.R` files in that directory. These files generally contain functions, methods, and R objects.

```
# R/add.R
string_length <- function(strings) {
  stringr::str_length(strings)
}
```

2.1.4   Documentation

Documenting packages used to be notoriously complicated but thanks to the package roxygen2 it no longer is the case. The documentation of functions of the package (accessible with `?`) and datasets that comprise the package reside in separate files in the `man/` directory. These are `.Rd` files that use a custom syntax resembling LaTex. The roxygen package eases the creation of these files by turning special comments and tags in `.R` files into said `.Rd` files.

Special comments are a standard R comment `#` followed by an apostrophe `'`. The first sentence of the documentation is the title of the documentation file while the second is the description.

```
#' Strings Length
#'
#' Returns the number of characters in strings.
string_length <- function(strings) {
  stringr::str_length(strings)
}
```

There are a plethora of roxygen2 tags to further document different sections, below we use two different tags to document the parameters and give an example.

```
#' Strings Length
#'
#' Returns the number of characters in strings.
#'
#' @param strings A vector of character strings.
#'
#' @example string_length(c("hello", "world"))
string_length <- function(strings) {
```

```
  stringr::str_length(strings)
}
```

As well as generating documentation the roxygen2 package also allows populating the `NAMESPACE` file. This is an extensive and often confusing topic but for the purpose of this book we'll content with the following: the `NAMESPACE` includes functions that are *imported* and *exported* by the package.

By default functions that are present in the R files in the `R/` directory are not exported: they are not accessible outside the package. Therefore the `string_length` function defined previously will not be made available to users of the package, only other function within the package will be able to call it. In order to export it we can use the `@export` tag. This will place the function as exported in the `NAMESPACE` file.

```
#' Strings Length
#'
#' Returns the number of characters in strings.
#'
#' @param strings A vector of character strings.
#'
#' @example string_length(c("hello", "world"))
#'
#' @export
string_length <- function(strings) {
  stringr::str_length(strings)
}
```

There are two ways to use external functions (functions from other R packages), as done thus far in the `string_length` function by using the namespace (package name) to call the function: `stringr::str_length`. Or by importing the function needed using a roxygen2 tag thereby removing the need for using the namespace.

```
#' Strings Length
#'
#' Returns the number of characters in strings.
#'
#' @param strings A vector of character strings.
#'
#' @example string_length(c("hello", "world"))
#'
#' @importFrom stringr str_length
```

```
#'
#' @export
string_length <- function(strings) {
  str_length(strings) # namespace removed
}
```

Above we import the function `str_length` from the `stringr` package using the `importFrom` roxygen2 tag. The first term following the tag is the name of the package wherefrom to import the functions and the following terms are the name of the functions separated by spaces so one can import multiple functions from the same package with, e.g.: `@importFrom stringr str_length str_to_upper`. If the package uses very many functions from a single package one might also consider importing said package in its entirety with e.g.: `@import stringr`.

Finally, one can actually generate the `.Rd` documentation files and populate the `NAMESPACE` with either the `devtools::document()` function or `roxygen2::roxygenise()`.

Remember to run `devtools::document()` after changing roxygen2 tags otherwise changes are not actually reflected in the `NAMESPACE` and documentation.

2.1.5 Installed files

Here we tackle the topic of installed files as it will be relevant to much of what the book covers. Installed files are files that are downloaded and copied as-is when users install the package. This directory will therefore come in very handy to store JavaScript files that package will require. These files can be accessed with the `system.file` function which will look for a file from the root of the `inst/` directory.

```
# return path to `inst/dependency.js` in `myPackage`
path <- system.file("dependency.js", package = "myPackage")
```

2.1.6 Build, load, and install

Finally, after generating the documentation of the package with `devtools::document()` one can install it locally with `devtools::install()`. This however can take a few seconds too many whilst developing a package as one iterates and regularly tries things; `devtools::load_all()` will not install the package but load all the functions and object in the global environment to let you run them.

There is some cyclical nature to developing packages:

1. Write some code
2. Run `devtools::document()` (if documentation tags have changed)
3. Run `devtools::load_all()`
4. Repeat

Note whilst this short guide will help you develop packages good enough for your system it will certainly not pass CRAN checks.

2.2   JSON

JSON (JavaScript Object Notation) is a very popular data *interchange* format with which we will work extensively throughout this book, it is thus crucial that we have a good understanding of it before we plunge into the nitty-gritty. As one might foresee, if we want two languages to work together it is essential that we have a data format that can be understood by both—JSON lets us harmoniously pass data from one to the other. While it is natively supported in JavaScript, it can be graciously handled in R with the jsonlite package[6] (**?**), in fact it is the serialiser used internally by all the packages we shall explore in this book.

"To serialise" is just jargon for converting data to JSON.

2.2.1   Serialising

JSON is to all intents and purposes the equivalent of lists in R; a flexible data format that can store pretty much anything–expect data.frames a structure that does not exist in JavaScript. 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).

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

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

---

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

```
  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.

```
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 the 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. Note that this only applies to vectors, lists of length one will be serialised to arrays of length one even with `auto_unbox` turned on: `list("hello")` will always be converted to `["hello"]`.

2.2.2   Tabular data

If JSON is more or less the equivalent of lists in R one might wonder how jsonlite handles dataframes since they do not exist in JavaScript.

```r
# 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. Another great example is to use `console.table` in the JavaScript console (more on that later) to display the table JSON as a table.

We can reproduce this with the snippet below, we remove row names and use apply to turn every row into a list.

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

> console.table([{"speed":4,"dist":2},{"speed":4,"dist":10},{"speed":7,"dist":4},{"speed":7,"dist":22},{"speed":8,"dist":16}])

VM1260:1

(index)	speed	dist
0	4	2
1	4	10
2	7	4
3	7	22
4	8	16

▶ Array(5)

FIGURE 2.2: console.table output

```
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.

2.3   JavaScript

The book is not meant to teach one JavaScript, only to show how graciously it can work with R. Let us thus go through the very basics to ensure we know enough to get started with the coming chapters.

The easiest way to run JavaScript interactively is probably to create an HTML file (e.g.: `try.html`), write your code within a `<script>` tags and open the file in your web browser. The console output can be observed in the console of the browser, developer tools (more on that below).

```html
<!-- index.html -->
<html>
  <head>
  </head>
  <body>
    <p id="content">Trying JavaScript!</p>
  </body>
  <script>
    // place your JavaScript code here
    console.log('Hello JavaScript!')
  </script>
</html>
```
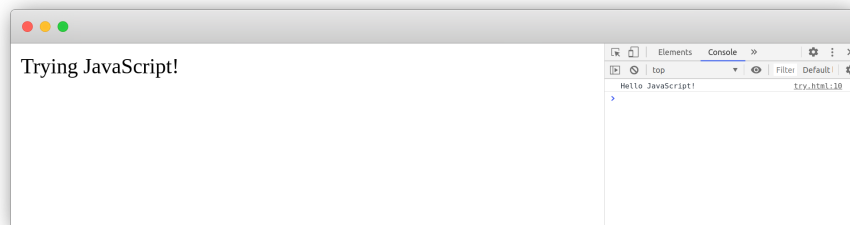
2.3.1   Developer tools

Most of the JavaScript code written in this book is intended to be run in web browsers, it is thus vital that you have a great understanding of your web browser and its developer tools (devtools). In this section we discuss those available in Google Chrome and Chromium but such tools, albeit somewhat different, also exist in Mozilla Firefox.

> The RStudio IDE is built on Chromium, some of these tools will therefore also work in RStudio.

The easiest way to access the developer the developer tools from the browser is by "inspecting:" right click on an element on a webpage and select "inspect". This will open the developer tools either at the bottom or on the right of the page.
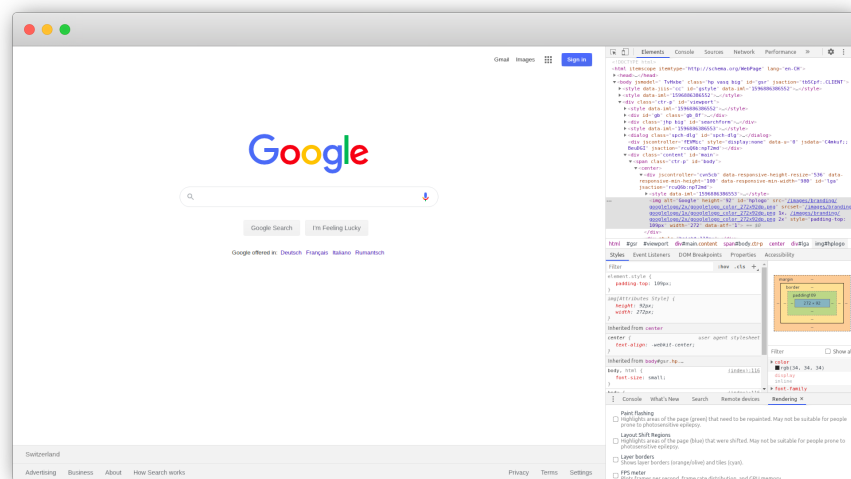

FIGURE 2.3: Google Chrome Devtools

The developer tools pane consists of several tabs but we will mainly use:

1. Elements: Presents the DOM Tree, the HTML document structure, great for inspecting the structure of the outputs generated from R.
2. Console: The JavaScript console where messages, errors, and other such things are logged. Essential for debugging.

2.3.2   Variable declaration and scope

One major way JavaScript differs from R is that variables must be declared
using one of three keywords, `var`, `let`, or `const` which mainly affect the scope
where the declared variable will be accessible from.

```
x = 1; // error
var x = 1; // works
```

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

```
var y; // declare
y = [1,2,3]; // define it as array
y = 'string'; // change to character string
```

The `let` and `const` keywords were added in ES2015, the `const` is used to
defined a constant: a variable that once declared cannot be changed.

```
const x = 1; // declare constant
x = 2; // error
```

Though this is probably only rarely done in R, one can produce something
similar by locking the binding for a variable in its environment.

```
x <- 1 # declare x
lockBinding("x", env = .GlobalEnv) # make constant
x <- 2 # error
```

```
## Error in eval(expr, envir, enclos): cannot change value of locked binding for 'x'
```

```
unlockBinding("x", env = .GlobalEnv) # unlock binding
x <- 2 # works
```

The `let` keyword is akin to declaring a variable with the `var` keyword, however
`let` (and `const`) will declare the variable in the "block scope." This in effect
further narrows down the scope where the variable will be accessible, a block
scope is generally the area within `if`, `switch` conditions or `for` and `while`
loops: areas within curly brackets.

```
if(true){
  let x = 1;
```

```
  var y = 1;
}

console.log(x) // error x does not exist
console.log(y) // works
```

In the above example `x` is only accessible within the if statement as it is declared with `let`, `var` does not have a block scope.

While on the subject of scope, 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 is extremely 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); // print to console
}

foo();
```

Accessing variables from the parent environment (context) is useful in JavaScript but should not be done in R

2.3.3   Document object model

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 the page only contains "Trying JavaScript!" when the page is opened (loaded) in the web browser JavaScript runs the code and changes it: this happens very fast so the original text cannot be seen.

```html
 <!-- 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>
```

One final thing to note for future reference, though not limited to the ids or classes most such selection of elements from the DOM are done with those where the pound sign refers to an element's id (`#id`) and a dot refers to an element's class (`.class`), just like in CSS.

```html
 <!-- index.html -->
<html>
  <head>
  </head>
  <body>
    <p id="content" class="stuff">Trying JavaScript!</p>
  </body>
  <script>
    // select by id
    var x = document.getElementById("content");
    var y = document.querySelector("#content");

    console.log(x == y); // true

    // select by class
    var z = document.querySelector(".stuff");
```

```
  </script>
</html>
```

Getting elements from the DOM is a very common operation in JavaScript. While a class can be applied to multiple elements which is useful to get and apply actions to multiple elements, the id attribute must be unique (no two elements can bear the same id in the HTML document) is useful to retrieve a specific element.

Interestingly some of that mechanism is used by shiny to retrieve and manipulate inputs, the argument `inputId` of shiny inputs effectively define the HTML `id` attribute of said input. Shiny can then internally make use of functions the likes of `getElementById` in order to get those inputs, set or update their values, etc.

```
shiny::actionButton(inputId = "theId", label = "the label")
```

the label

This of course only scratches the surface of JavaScript thus this 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 view the rendered source code.

Part II

Data Visualisation

3

Introduction to widgets

This part of the book explores 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 parts of this book we mainly learn through examples, building multiple widgets of increasing complexity as we progress through the chapter. Before writing the first widget, we explore existing R packages that allow creating interactive data visualisations this give a first glimpse at we build in this part of the book. Then 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 htmlwidgets work with shiny.

3.1 Plotly package

3.2 DT package

4

Basics of building widgets

Having explored existing packages that build on top of the htmlwidgets package gives some idea of the end product we learn to build but it does not give insights about where to start and much of how it works probably remains somewhat magical.

4.1 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 clients are JavaScript visualisation libraries let us take a look at some such popular libraries and briefly analyse at how they work and what they have in common. This will greatly help conceptualise what one is trying to achieve in this chapter.

4.1.1 Plotly

Plotly.js[1] is probably one of the more popular out there, it provides over 40 fully customiseable 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 the visualisation will be placed, then, using `Plotly.newPlot`, create the actual visualisation by passing it first the element previously mentioned and a JSON of options that describe the chart.

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

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

---

[1]https://plotly.com/javascript/

```html
</head>

<body>
  <!-- div to hold visualisation -->
  <div id="chart" style="width:600px;height:400px;"></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>
```
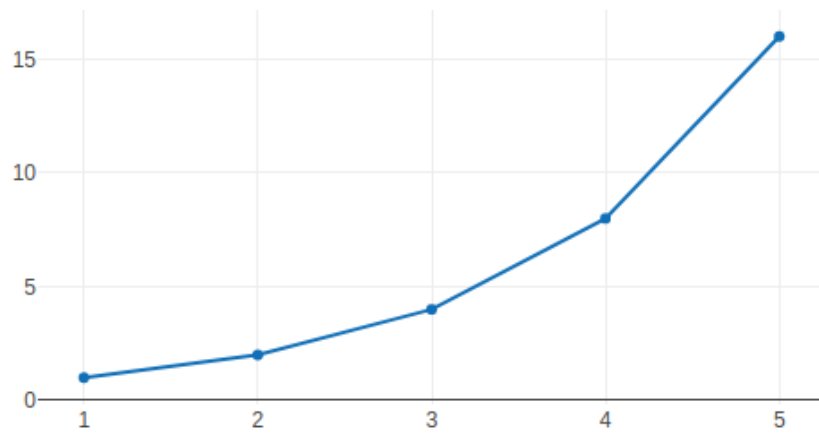

FIGURE 4.1: Plotly.js example

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

4.1.2   Highchart.js

Highcharts[2] is another library which allows creating gorgeous visualisation, maps, and more, it's also very popular albeit not being entirely open-source.

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

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

<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.
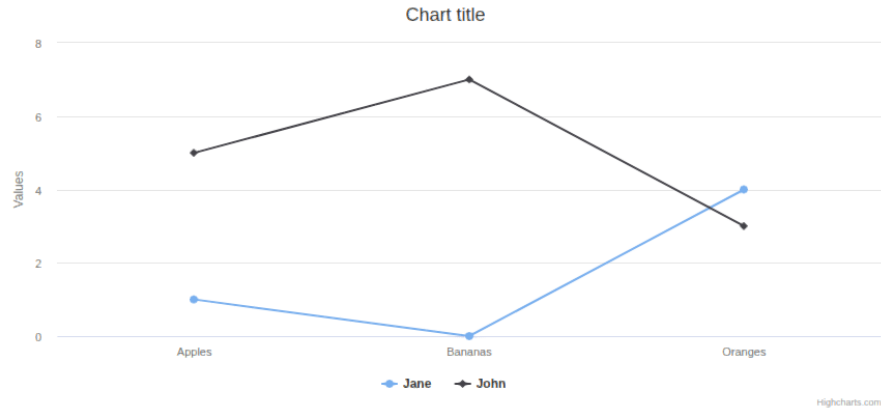
---

[2]https://www.highcharts.com/

FIGURE 4.2: Highcharts example

4.1.3   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>

<body>
  <!-- canvas to hold visualisation -->
  <canvas id="chart"></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',
```

---

[3]https://www.chartjs.org/

```
        data: [12, 19, 3, 5, 2, 3]
      }]
    }
  });
</script>
</body>

</html>
```
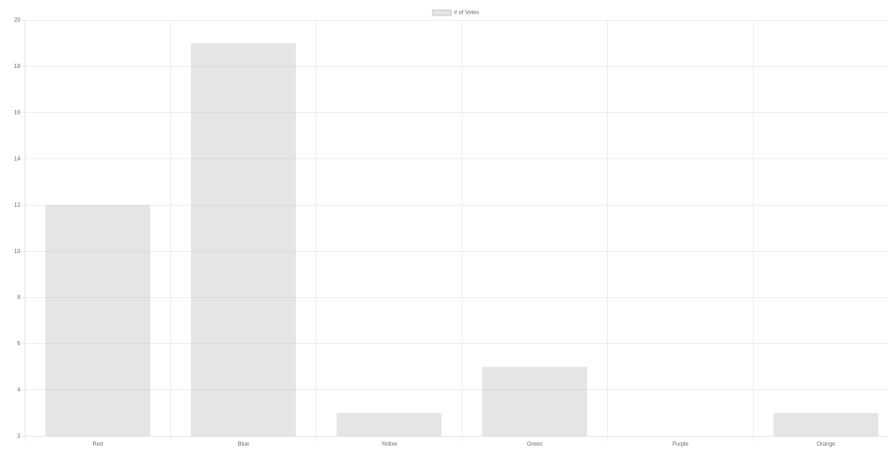

FIGURE 4.3: Chart.js example

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 as well as demonstrate how little JavaScript code is actually involved. It also hints at what should be reproduced, to some extent at least, using R.

4.2 How it works

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

As observed, 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 exter-
nal 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 visuali-
sation. Finally all should be managed to work seamlessly across R markdown,
shiny, and other environments.
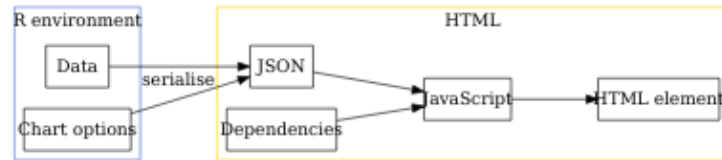
Thankfully the htmlwidgets package is there to handle most of this. Nonethe-
less, it is important to understand that these operations are undertaken (to
some degree) by htmlwidgets.

5

Your First Widget

The previous chapter gave some indication as to how widgets work but this is overall probably still shrouded in mystery. This chapter aims at demystifying what remains confusing. This is done by building a very basic widget with the aim of rummaging through its components to observe how they interact and ultimately grasp a greater understanding of how such interactive outputs are actually produced.

5.1 The Scaffold

Though one could probably create widgets outside of an R package, it would only make things more complicated, htmlwidgets naturally take the form of R packages and are stunningly simple to create. Below we create a package named "playground" which will be used to mess around and explore.

```
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.

> You can scaffold multiple widgets in a single package.

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 was created contains a function named "play" which takes, amongst other arguments, a message.

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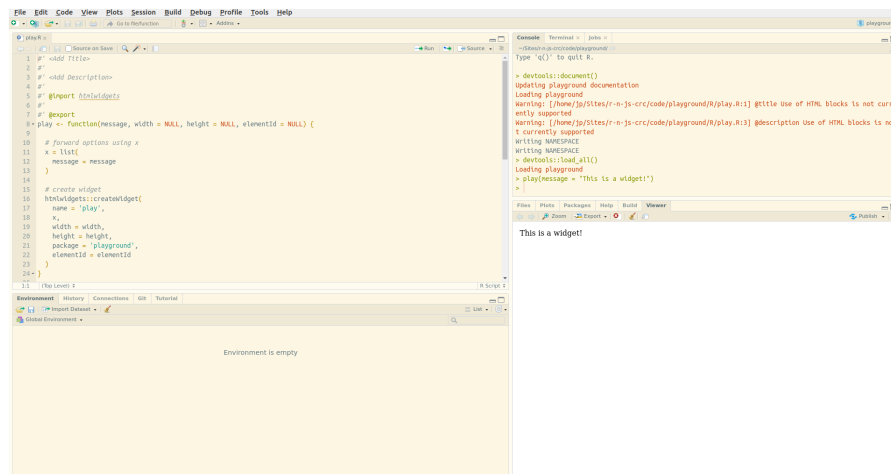

FIGURE 5.1: First HTML widget

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 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.

5.2 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`.

```r
# 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.

```html
<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.

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

```html
<!-- 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.

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

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

```html
<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.

5.3   JavaScript-side

Peaking inside the `play.js` file located at `inst/htmlwidgets/play.js` reveals the code below:

```js
// 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 intimate you. The name of the widget (`play`) corresponds to the name used when generating the scaffold, it can also be seen in the `createWidget` function used inside the `play.R` file.

```
htmlwidgets::createWidget(
  name = 'play',
  x,
  width = width,
  height = height,
  package = 'playground',
  elementId = elementId
)
```

This is so htmlwidgets can internally match the output of `createWidget` to its JavaScript function.

The `factory` function returns two functions, `resize`, and `renderValue`. The first is used to dynamically resize the output, it is not relevant to this widget is thus tackled later on. Let us focus on `renderValue`, the function that actually renders the output. It takes an object `x` from which the "message" variable is used as text for object `el` (`el.innerText`). The object `x` passed to this function is actually the list of the same name that was build 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
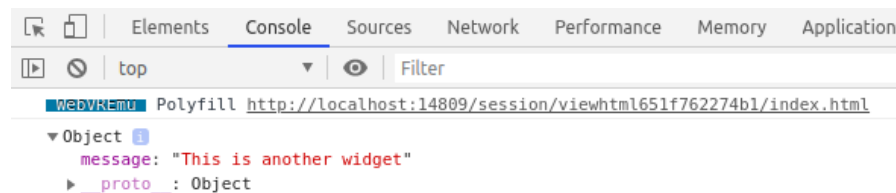explore the console from the browser (inspect and go to the "console" tab).

FIGURE 5.2: Console tab output

This displays the JSON object containing the message: it looks eerily sim-
ilar 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 that 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 understand-
ing of the el object, which can also be logged in the console.

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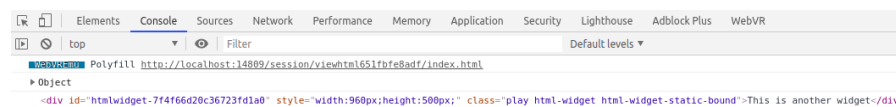

FIGURE 5.3: Console tab output

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 within it. 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.

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

class(html)
```

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

```r
# returns string
as.character(html)
```

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

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

```r
# 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.

```r
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.

6

A Realistic Widget

In this section we build a package called `typed`, which wraps the JavaScript library of the same name, typed.js[1] 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[2] to see how typed.js works.

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

<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>
```

---

[1]https://github.com/mattboldt/typed.js/
[2]https://github.com/mattboldt/typed.js/

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.

6.1 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[3] 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")
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 `.yml` file located at `inst/htmlwidgets` which at first contains a commented template.

---

[3]https://github.com/mattboldt/typed.js

```
# (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>
...
```

6.2   JavaScript

On its official website[4], 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) {

  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 is 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`)

---

[4]https://mattboldt.com/demos/typed-js/

```
// 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, including the text being typed is predefined: the package is currently not making any use the of the inputs passed from R. This can be changed by using `x.message` passed from R instead of the default strings.

```
// 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 and array (vector) and not a single string.

```
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.

```r
# 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.

```js
// 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.

```r
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
  )
}
```

6.3 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 `<span>`, 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(...)
}
```

Note that this can also be used to force certain attributes onto the element. For instance we could use the code below to force all output to use a red font.

```
typed_html <- function(...){
  htmltools::tags$span(..., style = "color:red;")
}
```

We shall leave it at that and move on to building another widget but completing the package makes for an interesting exercise.

7

The Full Monty

With a first widget built one can jump onto another one: gio.js[1], a library to draw arcs between countries on a 3 dimensional globe. This will include many more functionalities such packages can comprise.

Then again, the first order of business when looking to integrate a library is to look at the documentation to understand what should be reproduced in R.

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

<head>
  <!-- Import libraries -->
  <script src="three.min.js"></script>
  <script src="gio.min.js"></script>
</head>

<body>
  <!-- div to hold visualisation -->
  <div id="globe" style="width: 200px; height: 200px"></div>

  <!-- Script to create visualisation -->
  <script>
    var container = document.getElementById("globe");
    var controller = new GIO.Controller(container);
    controller.addData(data);
    controller.init();
  </script>
</body>

</html>
```

Gio.js has itself a dependency, three.js[2], which needs to be imported before

---

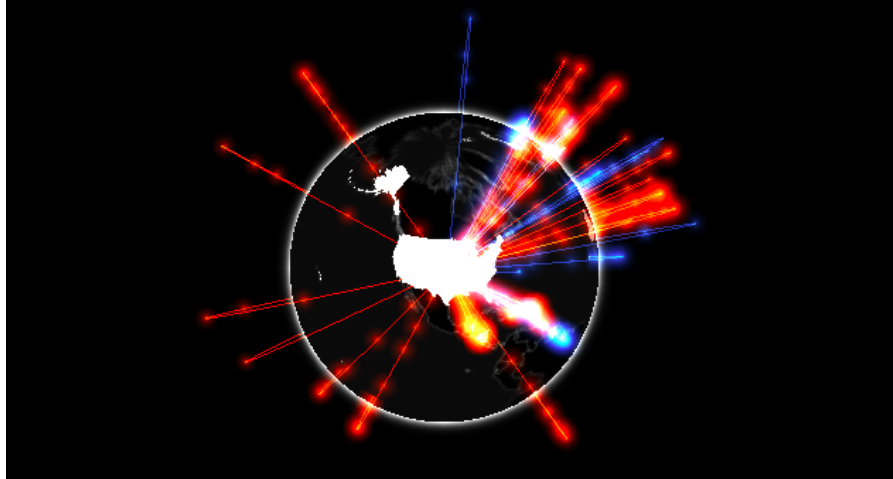[1]https://giojs.org/
[2]https://threejs.org/

FIGURE 7.1: Example of Gio.js visualisation

gio.js, other than that not much differs from libraries previously explored in
this chapter.

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

7.1 Dependencies

Handling the dependencies does not differ much, we only need to download
two libraries instead of one.

```
# create directories for JS dependencies
dir.create("./inst/htmlwidgets/three", recursive = TRUE)
dir.create("./inst/htmlwidgets/gio", recursive = TRUE)

# download JS dependencies
three <- "https://cdnjs.cloudflare.com/ajax/libs/three.js/110/three.min.js"
gio <- "https://raw.githubusercontent.com/syt123450/giojs/master/build/gio.min.js"

download.file(three, "./inst/htmlwidgets/three/three.min.js")
download.file(gio, "./inst/htmlwidgets/gio/gio.min.js")
```

This should produce the following working directory.

```
.
  DESCRIPTION
  NAMESPACE
  R
      gio.R
  inst
      htmlwidgets
          gio
              gio.min.js
          gio.js
          gio.yaml
          three
              three.min.js
```

The libraries have been downloaded but the `gio.yml` file is yet to be edited. The order in which the libraries are listed matters; just as in HTML three.js needs to precede gio.js as the latter depends on the former and not vice versa.

```yaml
dependencies:
  - name: three
    version: 110
    src: htmlwidgets/three
    script: three.min.js
  - name: gio
    version: 2.0
    src: htmlwidgets/gio
    script: gio.min.js
```

7.2 JavaScript

Let's copy the JavaScript code from the Get Started section of gio.js[3] in the `gio.js` file's `renderValue` function. At this point the data format is not known so we comment the line which adds data to the visualisation.

```javascript
// gio.js
HTMLWidgets.widget({
```

---

[3]https://giojs.org/index.html

```
  name: 'gio',

  type: 'output',

  factory: function(el, width, height) {

    // TODO: define shared variables for this instance

    return {

      renderValue: function(x) {

        var container = document.getElementById("globe");
        var controller = new GIO.Controller(container);
        //controller.addData(data);
        controller.init();

      },

      resize: function(width, height) {

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

      }

    };
  }
});
```

One can document and load the package but it will not work as the code above attempts to place the visualisation in a div with id = "globe". As for the previously written widget, this needs to be changed to el.id.

```
// gio.js
renderValue: function(x) {

  var container = document.getElementById(el.id);
  var controller = new GIO.Controller(container);
  //controller.addData(data);
  controller.init();

}
```

At this stage the widget should generate a visualisation.

```
devtools::document()
devtools::load_all()
gio(message = "This required but not used")
```


FIGURE 7.2: Output without data

Not too shabby given how little work was put into this! Before we move on let us optimise something. In the JavaScript code we retrieve the `container` using `el.id` but this in effect is very inefficient: `el` is identical to `container`.

```
// gio.js
renderValue: function(x) {

  var controller = new GIO.Controller(el);
  //controller.addData(data);
  controller.init();

}
```

7.3 Working with Data

An interesting start, now onto adding data. Let's take a look at the documentation[4] to see what data the library expects.

---

[4] https://giojs.org/html/docs/dataAdd.html

```
[
  {
    "e": "CN",
    "i": "US",
    "v": 3300000
  },
  {
    "e": "CN",
    "i": "RU",
    "v": 10000
  }
]
```

The JSON data should constitutes of arrays that denote arcs to draw on the globe where each arc is defined by an exporting country (`e`), an importing country (`i`), and is given a value (`v`). The importing and exporting country, the source and target of the arc, are indicated by ISO alpha-2 country codes. We can read this JSON into R.

```
# data.frame to test
arcs <- jsonlite::fromJSON(
  '[
    {
      "e": "CN",
      "i": "US",
      "v": 3300000
    },
    {
      "e": "CN",
      "i": "RU",
      "v": 10000
    }
  ]'
)

print(arcs)
```

```
##    e  i        v
## 1 CN US 3300000
## 2 CN RU    10000
```

Jsonlite automagically converts the JSON into a data frame where each row is an arc, which is great as R users tend to prefer rectangular data over lists:

this is probably what the package should use as input too. Let us make some changes to the `gio` function so it takes data as input.

```r
gio <- function(data, width = NULL, height = NULL, elementId = NULL) {

  # forward options using x
  x = list(
    data = data
  )

  # create widget
  htmlwidgets::createWidget(
    name = 'gio',
    x,
    width = width,
    height = height,
    package = 'gio',
    elementId = elementId
  )
}
```

This must be reflected in the `play.js` file where we uncomment the line previously commented and use `x.data` passed from R.

```js
// gio.js
renderValue: function(x) {

  var controller = new GIO.Controller(el);
  controller.addData(x.data); // uncomment & use x.data
  controller.init();

}
```

We can now use the function with the data to plot arcs!

```r
devtools::document()
devtools::load_all()
gio(arcs)
```

Unfortunately, this breaks everything and we are presented with a blank screen. Using `console.log` or looking at the source code the rendered widget reveals the problem: the data isn't actually in the correct format!

```
{"x":{"data":{"e":["CN","CN"],"i":["US","RU"],"v":[3300000,10000]}},"evals":[],"jsHooks":[
```

Htmlwidgets actually serialised the data frame column-wise (long) where each
array is a column, whereas gio.js expect the data to be wide (row-wise serial-
isation) where each array is a arc (row).

```r
# column-wise
jsonlite::toJSON(arcs, dataframe = "columns")
```

```
## {"e":["CN","CN"],"i":["US","RU"],"v":[3300000,10000]}
```

```r
# row-wise
jsonlite::toJSON(arcs, dataframe = "rows")
```

```
## [{"e":"CN","i":"US","v":3300000},{"e":"CN","i":"RU","v":10000}]
```

As mentioned previously, convention has it that rectangular data (data frames)
are serialised row-wise. This is likely to be a recurring problem.

7.4 Transforming Data

There are multiple ways to transform the data and ensure the serialised JSON
is as the JavaScript library expects it to be.

7.4.1 In JavaScript

JavaScript can be used to transform the data, leaving the serialiser as-is
to reshape the data in the client. The HTMLwidget JavaScript library (al-
ready imported by default) exports an object which provides a method,
`dataframeToD3`, to transform the data from long to wide.

```js
// gio.js
renderValue: function(x) {

  // long to wide
  x.data = HTMLWidgets.dataframeToD3(x.data);

  var controller = new GIO.Controller(el);
  controller.addData(x.data);
  controller.init();
```

```
}
```

7.4.2   In R

Instead of serialising the data a certain way then correct in JavaScript as demonstrated previously, one can also modify, or even replace, htmlwidgets' default serialiser. Speaking of which, below is the default serializer used by htmlwidgets.

```
function (x, ..., dataframe = "columns", null = "null", na = "null",
  auto_unbox = TRUE, digits = getOption("shiny.json.digits", 16),
  use_signif = TRUE, force = TRUE, POSIXt = "ISO8601", UTC = TRUE,
  rownames = FALSE, keep_vec_names = TRUE, strict_atomic = TRUE)
{
  if (strict_atomic)
      x <- I(x)
  jsonlite::toJSON(x, dataframe = dataframe, null = null, na = na,
    auto_unbox = auto_unbox, digits = digits, use_signif = use_signif,
    force = force, POSIXt = POSIXt, UTC = UTC, rownames = rownames,
    keep_vec_names = keep_vec_names, json_verbatim = TRUE, ...)
}
```

The problem at hand is caused by the `dataframe` argument which is set to `columns` where it should be set `rows` (for row-wise). Arguments are passed to the serialiser indirectly, in the form of a list set as `TOJSON_ARGS` attribute to the object `x` that is serialised. We could thus change the `gio` function to reflect the aforementioned change.

```
gio <- function(data, width = NULL, height = NULL, elementId = NULL) {

  # forward options using x
  x = list(
    data = data
  )

  # serialise data.frames to wide (not long as default)
  attr(x, 'TOJSON_ARGS') <- list(dataframe = "rows")

  # create widget
  htmlwidgets::createWidget(
    name = 'gio',
    x,
```

```
      width = width,
      height = height,
      package = 'gio',
      elementId = elementId
  )
}
```

The above may appear confusing at first as one rarely encounters the `attr` replacement function.

```
attr(cars, "hello") <- "world" # set
attr(cars, "hello") # get
```

```
## [1] "world"
```

Otherwise the serializer can also be replaced in its entirety, also by setting an attribute, `TOJSON_FUNC`, to the `x` object. Below the serialiser is changed to jsonify (**?**) which by default serialises data frames to wide, unlike htmlwidgets' serialiser, thereby also fixing the issue.

```
gio <- function(data, width = NULL, height = NULL, elementId = NULL) {

  # forward options using x
  x = list(
    data = data
  )

  # replace serialiser
  attr(x, 'TOJSON_FUNC') <- gio_serialiser

  # create widget
  htmlwidgets::createWidget(
    name = 'gio',
    x,
    width = width,
    height = height,
    package = 'gio',
    elementId = elementId
  )
}

# serialiser
gio_serialiser <- function(x){
```

```
  jsonify::to_json(x, unbox = TRUE)
}
```

Alternatively, one can also leave all serialisers unchanged and instead format the data in R prior to the serialisation, changing the dataframe to a row-wise list.

```
x = list(
  data = apply(data, 1, as.list)
)
```

7.4.3 Pros & Cons

There are pros and cons to each methods. The preferable method is probably to only alter the argument(s) where needed, this is the method used in the remainder of the book. Replacing the serialiser in its entirety should not be necessary, only do this once you are very familiar with serialisation and truly see a need for it. Moreover htmlwidgets' serialiser extends jsonlite to allow converting JavaScript code which will come in handy later on. Transforming the data in JavaScript has one drawback, `HTMLWidgets.dataframeToD3` cannot be applied to the entire `x` object, it will only work on the subsets that hold the column-wise data (`x.data`) which tends to lead to clunky code as one uses said function in various places.
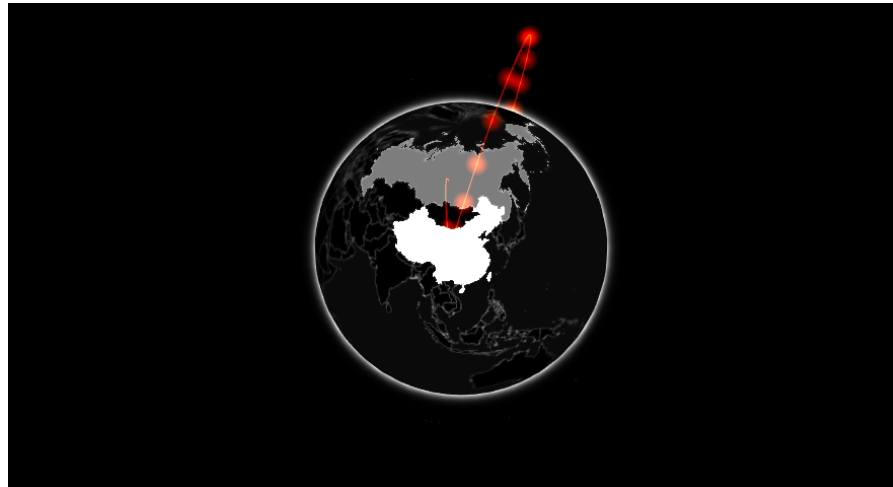

FIGURE 7.3: Gio output with correct serialisation

7.5   On Print

Let's add the option to style the globe, gio.js provides mutliple themes[5] but they are currently not applicable from R. As a matter of fact, gio.js provides dozens of customisation options that should be available in the package as well. These, however, probably should be split across different functions, just like they are in gio.js, rather than all be accessible from a single function containing hundreds of arguments. This begs the question, when would one use another function given the function `gio` generates the visualisation? As it happens `gio` itself (or rather the function `htmlwidgets::createWidget` it contains) does not render the output, it returns an object of class "htmlwidget" which actually renders the visualisation on print (literally `htmlwidget.print` method).

```r
g <- gio(arcs) # nothing renders
g # visualisation renders
```

Therefore, one can use functions on the object returned by `gio` which contains, amongst other things, the `x` list to which we can make changes, append data, or do any other operation that standard R lists allow.

```r
print(g$x)

## $data
##    e   i       v
## 1 CN US 3300000
## 2 CN RU   10000
##
## attr(,"TOJSON_ARGS")
## attr(,"TOJSON_ARGS")$dataframe
## [1] "rows"
```

This clarified the function to change the style of the visualisation can be added to the package. It would take as input the output of `gio` and append the style (name of theme) to the `x` list, this would then be used in JavaScript to change the look of the visualisation.

```r
#' @export
gio_style <- function(g, style = "magic"){
```

---

[5]https://giojs.org/html/docs/colorStyle.html

```r
  g$x$style <- style
  return(g)
}
```

The style is applied using the `setStyle` method on the controller before the visualisation is created, before the `init` method is called, using the style passed from R: `x.style`.

```js
// gio.js
renderValue: function(x) {

  var controller = new GIO.Controller(el);
  controller.addData(x.data);

  controller.setStyle(x.style); // set style

  controller.init();

}
```

We can now run `devtools::load_all` to export the newly written function and load the functions in the environment with `devtools::load_all`.

```r
g1 <- gio(arcs)
g2 <- gio_style(g1, "juicyCake")

g2
```

This is great but can be greatly improved upon with the magrittr pipe (**?**), it would allow easily passing the output of each function to the next to obtain an API akin to that of plotly or highcharter.

```r
library(magrittr)

gio(arcs) %>%
  gio_style("juicyCake")
```

The pipe drastically improves the API that gio provides its users and thus probably should be exported by the package; the usethis package provides a function to easily do so.

FIGURE 7.4: Gio with a new style

```
# export the pipe
usethis::use_pipe()
```

This closes this chapter but is not the last we see of gio.js, we shall use it as example in the next chapters as we extend its functionalities, polish certain aspects such as sizing and security.

8

Advanced Topics

In the previous chapter we put together an interesting, fully functioning widget but it lacks polish and does not use all the features htmlwidgets provides, this chapter explores those. We look into handling the size of widgets to ensure they are responsive as well as discuss potential security concerns and how to address them. Finally we show how to pass JavaScript code from R to JavaScript and how to add HTML content before and after the widget.

8.1 Shared Variables

Up until now the topic of shared variables had been omitted as it was not relevant, however, it will be from here onwards. Indeed we are about to discover how to further manipulate the widget; changing the data, resizing, and more. This will generally involve the JavaScript instance of the visualisation, the object named `controller` in the gio package, which, being defined in the `renderValue` function, is not accessible outside of it. To make it accessible outside of `renderValue` requires a tiny but consequential change without which resizing the widget will not be doable for instance.

The `controller` variable has to be declared outside of the `renderValue` function, inside the `factory`. This was in fact indicated from the onset by the following comment: `// TODO: define shared variables for this instance` (generated by `htmlwidgets::scaffoldWidget`). Any variable declared as shown below will be accessible by all functions declared in the `factory`; `renderValue`, but also `resize` and others yet to be added.

```
HTMLWidgets.widget({

  name: 'gio',

  type: 'output',

  factory: function(el, width, height) {
```

```
    // TODO: define shared variables for this instance
    var controller;

    return {

      renderValue: function(x) {

        controller = new GIO.Controller(el); // declared outside

        // add data
        controller.addData(x.data);

        // define style
        controller.setStyle(x.style);

        // render
        controller.init();

      },

      resize: function(width, height) {

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

      }

    };
  }
});
```

8.1.1   Sizing

The `gio` function of the package we developed in the previous chapter has
arguments to specify the dimensions of the visualisation (width and height).
However, think how rarely (if ever) one specifies these parameters when using
plotly, highcharter or leaflet. Indeed HTML visualisations should be respon-
sive and fit the container they are placed in—not to be confused though,
these are two different things. This enables creating visualisations that look
great on large desktop screens as well as the smaller mobile phones or iPad
screens. Pre-defining the dimensions of the visualisation (e.g.: `400px`), breaks
all responsiveness as the width is no longer relative to its container. Using a
relative width like `100%` ensures the visualisation always fits in the container
edge to edge and enables responsiveness.

```r
arcs <- jsonlite::fromJSON(
  '[
    {
      "e": "CN",
      "i": "US",
      "v": 3300000
    },
    {
      "e": "CN",
      "i": "RU",
      "v": 10000
    }
  ]'
)

gio(arcs)
```
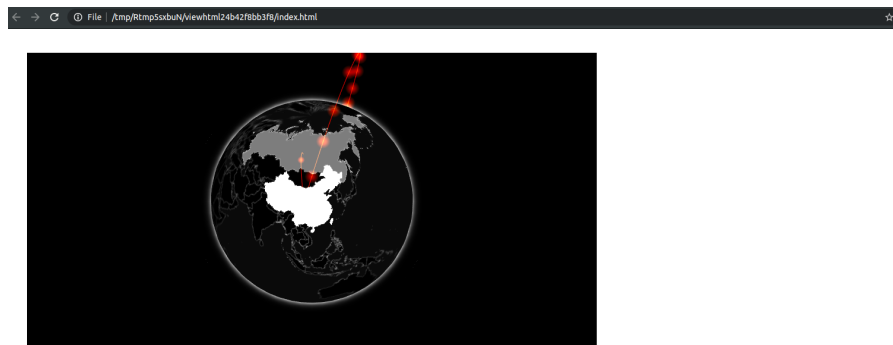

FIGURE 8.1: Gio with no sizing management

When this is not specified htmlwidgets sets the width of the visualisation to 400 pixels.

```r
gio(arcs, width = 500) # 500 pixels wide
gio(arcs, width = "100%") # fits width
```

These options are destined for the user of the package, the next section details how the developer can define default sizing behaviour.

8.1.2 Sizing Policy

One can specify a sizing policy when creating the widget, the sizing policy will dictate default dimensions and padding given different contexts:

- Global defaults
- RStudio viewer
- Web browser
- R markdown

It is often enough to specify general defaults as widgets are rarely expected to behave differently with respect to size depending on the context but it can be useful in some cases.

Below we modify the sizing policy of gio via the `sizingPolicy` argument of the `createWidget` function. The function `htmlwidgets::sizingPolicy` has many arguments, we set the default width to 100% to ensure the visualisation fills its container entirely regardless of where it is rendered. We also remove all padding by setting it to 0 and set `browser.fill` to `TRUE` so it automatically resizes the visualisation to fit the entire browser page.

```r
# create widget
htmlwidgets::createWidget(
  name = 'gio',
  x,
  width = width,
  height = height,
  package = 'gio',
  elementId = elementId,
  sizingPolicy = htmlwidgets::sizingPolicy(
    defaultWidth = "100%",
    padding = 0,
    browser.fill = TRUE
  )
)
```

8.2 Resizing

In the first widget built in this book (`playground`), when we deconstructed the JavaScript `factory` function but omitted the `resize` function. The `resize` function does what it says on the tin: it is called when the widget is resized. What this function will contain entirely depends on the JavaScript library one is working with. Some are very easy to resize other less so, that is for the
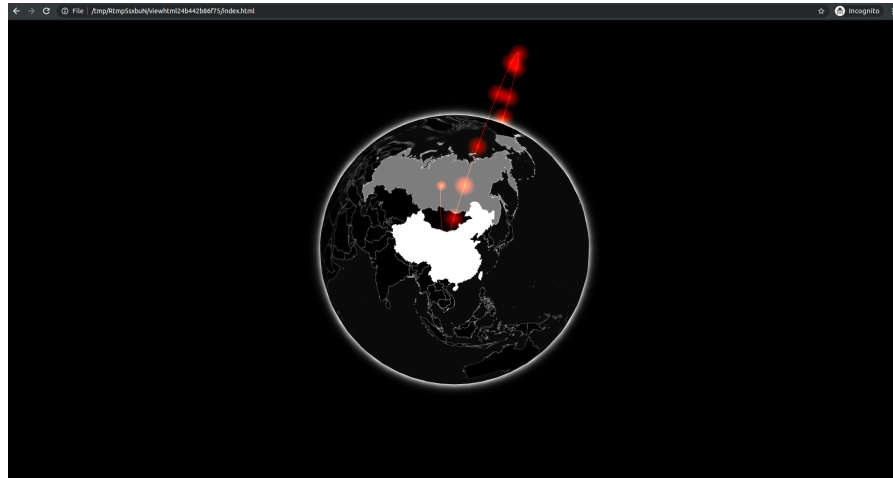
FIGURE 8.2: Gio with sizing policy

developer to discover in the documentation of the library. Some libraries, like gio, do not even require using a resizing function and handle that automatically under the hood; resize the width of the RStudio viewer or web browser and gio.js resizes too. This said, there is a function to force resize gio, though it is not in the official documentation it can be found in the source code: `resizeUpdate` is a method of the controller and does not take any argument.

```
...
resize: function(width, height) {
  controller.resizeUpdate();
}
...
```

To give the reader a better idea of what these tend to look like below are the ways plotly, highcharts, and chart.js do it.

Plotly

```
Plotly.relayout('chartid', {width: width, height: height});
```

Highcharts

```
chart.setSize(width, height);
```
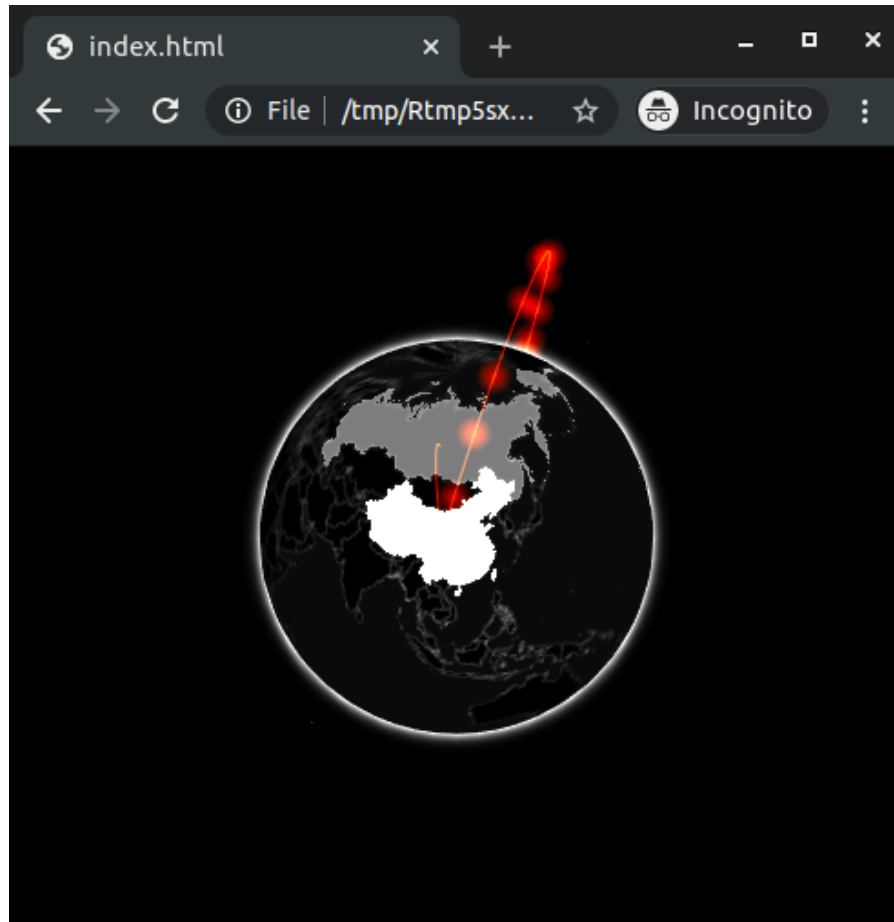
Chart.js

FIGURE 8.3: Gio resized

```
chart.resize();
```

Note that the `width` and `height` used in the functions above are obtained from the `resize` function itself (see arguments).

That is one of the reasons for ensuring the instance of the visualisation (`controller` in this case) is shared (declared in `factory`), if declared in the `renderValue` function then the `resize` function cannot access that object and thus cannot run the function required to resize the widget.

8.3 Pre Render Hooks & Security

The `createWidget` function also comes with a `preRenderHook` argument which accepts a function that is run just prior to the rendering, this function should accept the entire widget object as input, and should return a modified widget object. This was not used in any of the widgets previously built but is extremely useful. It can be used to make checks on the object to ensure all is correct, or remove variables that should only be used internally, and much more.

Currently, `gio` takes the data frame `data` and serialises it in its entirety which will cause security concerns as all the data used in the widget is visible in the source code of the output. What if the data used for the visualisation contained an additional column with sensitive information? We ought to ensure gio only serialises the data necessary to produce the visualisation.

```r
# add a variable that should not be shared
arcs$secret_id <- 1:2
```

We create a `render_gio` function which accepts the entire widget, filters only the column necessary from the data and returns the widget. This function is then passed to the argument `preRenderHook` of the `htmlwidgets::createWidget` call. This way, only the columns `e`, `v`, and `i` of the data are kept thus the `secret_id` column will not be exposed publicly.

```r
# preRenderHook function
render_gio <- function(g){
  # only keep relevant variables
  g$x$data <- g$x$data[,c("e", "v", "i")]
  return(g)
}

# create widget
htmlwidgets::createWidget(
  name = 'gio',
  x,
  width = width,
  height = height,
  package = 'gio',
  elementId = elementId,
  sizingPolicy = htmlwidgets::sizingPolicy(
    defaultWidth = "100%",
```

```
    padding = 0,
    browser.fill = TRUE
  ),
  preRenderHook = render_gio # pass renderer
)
```

Moreover, security aside, this can also improve performances as only the data relevant to the visualisation is serialised and subsequently loaded by the client. Without the modification above, were one to use `gio` on a dataset with 100 columns all would have been serialised, thereby greatly impacting performances both of the R process rendering the output and the web browser viewing the visualisation.

8.4   JavaScript Code

As mentioned in a previous chapter JavaScript code cannot be serialised to JSON.

```
# serialised as string
jsonlite::toJSON("var x = 3;")
```

```
## ["var x = 3;"]
```

Nonetheless, it is doable with htmlwidgets' serialiser (and only that one). The function `htmlwidgets::JS` can be used to mark a character vector so that it will be treated as JavaScript code when evaluated in the browser.

```
htmlwidgets::JS("var x = 3;")
```

```
## [1] "var x = 3;"
## attr(,"class")
## [1] "JS_EVAL"
```

This can be useful where the library requires the use of callback functions for instance.

Replacing the serialiser will break this feature.

8.5 Prepend & Append Content

There is the ability to append of prepend HTML content to the widget (shiny, htmltools tags, or a list of those). For instance, we could use `htmlwidgets::prependContent` to allow displaying a title to the visualisation as shown below.

```
#' @export
gio_title <- function(g, title){
  title <- htmltools::h3(title)
  htmlwidgets::prependContent(g, title)
}
```

```
gio(arcs) %>%
  gio_title("Gio.js htmlwidget!")
```

While the `prependContent` function places the content above the visualisation, the `appendContent` function places it below, as they accept any valid htmltools or shiny tag they can also be used for conditional CSS styling for instance.

`prependContent` and `appendContent` do not work in shiny.

8.6 Dependencies

Thus far, this book has only covered one of two ways dependencies can be included in htmlwidgets. Though the one covered, using the `.yml` file, will likely be necessary for every widget it has one drawback: all dependencies listed in the file are always included with the output. Dependencies can greatly affect the load time of the output (be it a standalone visualisation, an R markdown document, or a shiny application) as these files may be large. Most large visualisation libraries will therefore allow to bundle those dependencies in separate files. For instance, ECharts.js provides a way to customise the bundle to only include dependencies for charts that one wants to draw (e.g.: bar chart, or boxplot), Highcharts also allows splitting dependencies so one can load those needed for maps, stock charts, and more, separately. It is thus good practice to do the same in widgets so only the required dependencies are loaded, e.g.: when the user produces a map, only the dependency for that map is loaded. It is used in the leaflet package to load map tiles for instance.
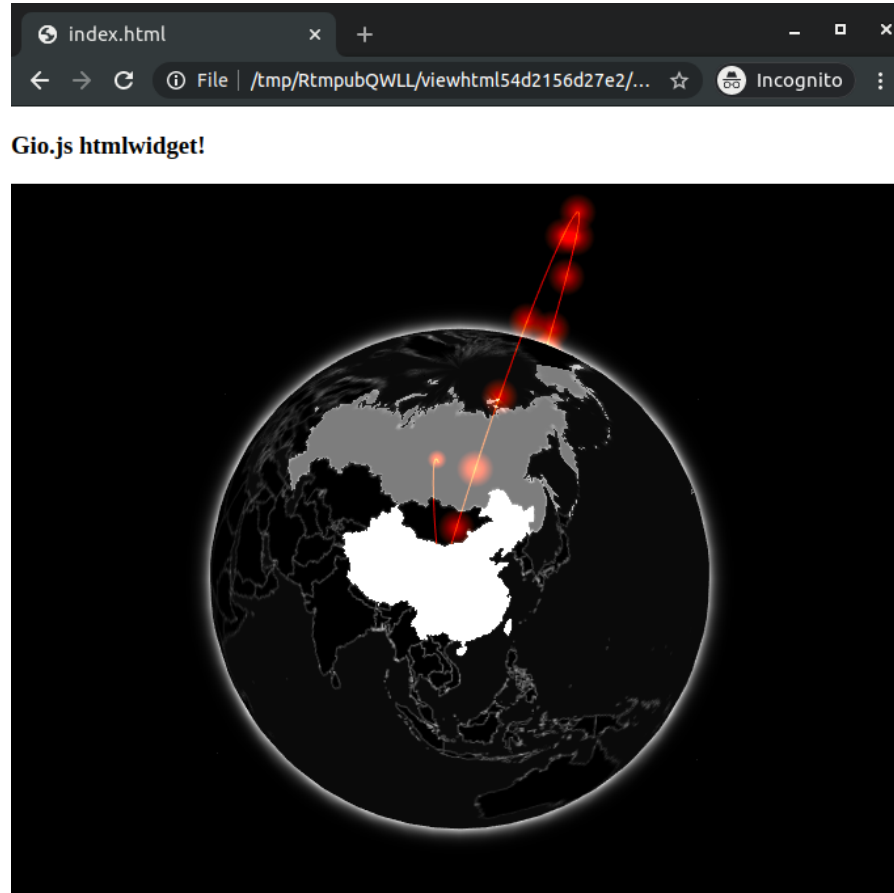
FIGURE 8.4: Gio output with a title

The Google Chrome network tab shows information on resources downloaded by the browser (including dependencies) including how long it takes. It is advisable to take a look at it to ensure no dependency drags load time.

To demonstrate, we will add a function in gio to optionally include stats.js[1], a JavaScript performance monitor which displays information such as the number of frames per second (FPS) rendered, or the number of milliseconds needed to render the visualisation. Gio.js natively supports stats.js but the dependency needs to be imported and that option needs to be enabled on the `controller` as shown in the documentation[2].

---

[1]https://github.com/mrdoob/stats.js/
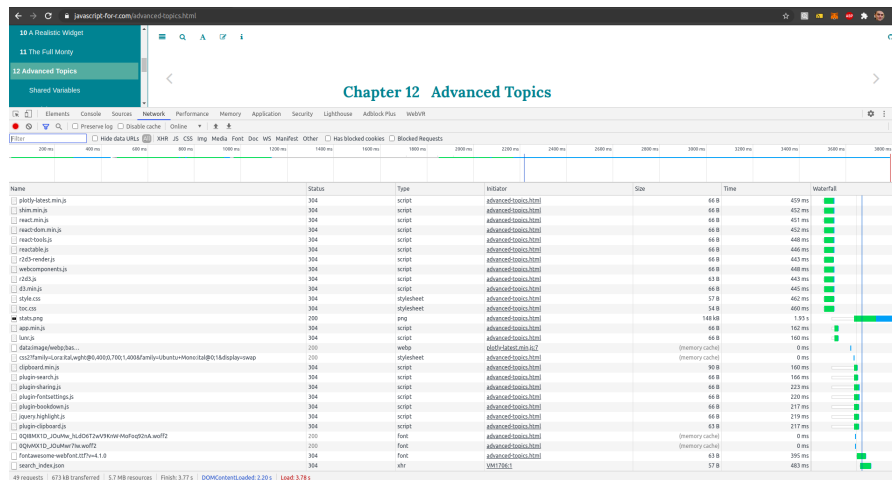[2]https://giojs.org/html/docs/interfaceStats.html

FIGURE 8.5: Google Chrome network tab

```
// enable stats
controller.enableStats();
```

In htmlwidgets those additional dependencies can be specified via the `dependencies` argument in the `htmlwidgets::createWidget` function, or they can be appended to the output of that function.

```
# create gio object
g <- gio::gio(arcs)

is.null(g$dependencies)
```

```
[1] TRUE
```

As shown above, the object created by `gio` includes dependencies, currently `NULL` as no such extra dependency is specified. One can therefore append those to that object in a fashion similar to what the `gio_style` function does.

From the root of the gio package we create a new directory for the stats.js dependency and download version `r17` from the CDN.

```
dir.create("htmlwidgets/stats")
url <- paste0(
  "https://cdnjs.cloudflare.com/ajax/libs/",
  "stats.js/r17/Stats.min.js"
)
download.file(url, destfile = "htmlwidgets/stats/stats.min.js")
```

First we use the `system.file` function to retrieve *the path to the directory* which contains the dependency (`stats.min.js`). It's important that it is the path to the directory and not the file itself.

```r
# stats.R
gio_stats <- function(g){

  # create dependency
  path <- system.file("htmlwidgets/stats", package = "gio")

  return(g)

}
```

Then we use the htmltools package to create a dependency, the `htmltools::htmlDependency` function returns an object of class `html_dependency` which htmlwidgets can understand and subsequently insert in the output. On the `src` parameter, since we reference a dependency from the filesystem we name the character string `file` but we could use the CDN (web hosted file) and name it `href` instead.

```r
# stats.R
gio_stats <- function(g){

  # create dependency
  path <- system.file("htmlwidgets/stats", package = "gio")
  dep <- htmltools::htmlDependency(
    name = "stats",
    version = "17",
    src = c(file = path),
    script = "stats.min.js"
  )

  return(g)

}
```

The dependency then needs to be appended to the htmlwidgets object.

```r
# stats.R
gio_stats <- function(g){

  # create dependency
  path <- system.file("htmlwidgets/stats", package = "gio")
```

```
  dep <- htmltools::htmlDependency(
    name = "stats",
    version = "17",
    src = c(file = path),
    script = "stats.min.js"
  )

  # append dependency
  g$dependencies <- append(g$dependencies, list(dep))

  return(g)

}
```

Finally, we pass an additional variable in the list of options (x) which we will use JavaScript-side to check whether stats.js must be enabled.

```
#' @export
gio_stats <- function(g){

  # create dependency
  path <- system.file("htmlwidgets/stats", package = "gio")
  dep <- htmltools::htmlDependency(
    name = "stats",
    version = "17",
    src = c(file = path),
    script = "stats.min.js"
  )

  # append dependency to gio.js
  g$dependencies <- append(g$dependencies, list(dep))

  # add stats variable
  g$x$stats <- TRUE

  return(g)
}
```

Then it's a matter of using the stats variable added to x in the JavaScript renderValue function to determine whether the stats feature should be enabled.

```js
// gio.js
if(x.stats)
  controller.enableStats();

controller.init();
```

Then the package can be documented to export the newly created function and loaded in the environment to test the feature.

```r
# create gio object
arcs %>%
  gio() %>%
  gio_stats()
```
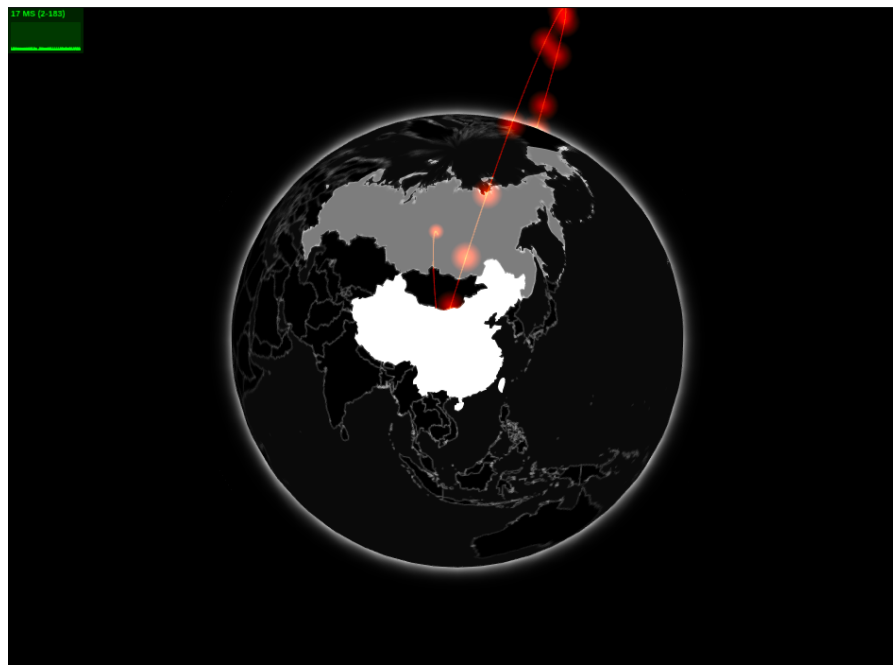

FIGURE 8.6: Gio with stats output

In brief, it is better to only place the hard dependencies in the `.yml` file; dependencies that are absolutely necessary to producing the visualisation and use dynamic dependencies where ever possible. Perhaps one can think of it as the difference between `Imports` and `Suggests` in an R package `DESCRIPTION` file.

8.7   Compatibility

One issue that might arise is that of compatibility between widgets. What if someone else builds another htmlwidget for gio.js which uses a different version of the library and that a user decides to use both packages in a shiny app or R markdown document? Something is likely to fail as two different version of gio.js are imported and that one overrides the other. For instance, the package echarts4r (**?**) allows working with leaflet but including the dependencies could clash with the leaflet package itself, therefore it uses the dependencies from the leaflet package instead.

The htmlwidgets package comes with a function to extract the dependencies from a widget, so they can be reused in another. The function `htmlwidgets::getDependency` returns a list of objects of class `html_dependency` which can therefore be used in other widgets as demonstrated in the previous section.

```
# get dependencies of the gio package
htmlwidgets::getDependency("gio")[2:3]
```

```
## [[1]]
## List of 10
##  $ name      : chr "three"
##  $ version   : chr "110"
##  $ src       :List of 1
##   ..$ file: chr "/home/jp/R/x86_64-pc-linux-gnu-library/4.0/gio/htmlwidgets/three"
##  $ meta      : NULL
##  $ script    : chr "three.min.js"
##  $ stylesheet: NULL
##  $ head      : NULL
##  $ attachment: NULL
##  $ package   : NULL
##  $ all_files : logi TRUE
##  - attr(*, "class")= chr "html_dependency"
##
## [[2]]
## List of 10
##  $ name      : chr "gio"
##  $ version   : chr "2"
##  $ src       :List of 1
##   ..$ file: chr "/home/jp/R/x86_64-pc-linux-gnu-library/4.0/gio/htmlwidgets/gio"
##  $ meta      : NULL
##  $ script    : chr "gio.min.js"
```

```
##   $ stylesheet: NULL
##   $ head      : NULL
##   $ attachment: NULL
##   $ package   : NULL
##   $ all_files : logi TRUE
##   - attr(*, "class")= chr "html_dependency"
```

8.8   Unit Tests

The best way to write unit tests for htmlwidgets is to test the object created by `htmlwidgets::createWidget`. Below we provide an example using testthat(**?**), running `expect*` functions on the output of `gio`.

```r
library(gio)
library(testthat)

test_that("gio has correct data", {
  g <- gio(arcs)

  # internally stored as data.frame
  expect_is(g$x$data, "data.frame")

  # gio does not work without data
  expect_error(gio())
})
```

8.9   Performances

A few hints have already been given to ensure one does not drain the browser, consider assessing the performances of the widget as it is being built. Always try and imagine what happens under the hood of the htmlwidget as you build it, it often reveals potential bottlenecks and solutions.

Remember that data passed to `htmlwidgets::createWidget` is 1) loaded into R, 2) serialised to JSON, 3) embedded into the HTML output, 4) read back in with JavaScript, which adds some overhead considering it might be read into JavaScript directly. This will not be a problem for most visualisations but might become one when that data is large. Indeed, there are sometimes

more efficient ways to load data into web browsers where it is needed for the visualisation.

Consider for instance geographic features (topoJSON and GeoJSON), why load them into R if it is to then re-serialise it to JSON?

Also keep the previous remark in mind when repeatedly serialising identical data objects, GeoJSON is again a good example. A map used twice or more should only be serialised once or better not at all. Consider providing other ways to for the developer to make potentially large data files accessible to the browser.

Below is an example of function that could be used within R markdown or shiny UI to load data in the front-end and bypass serialisation. Additionally the function makes use of AJAX (Asynchronous JavaScript And XML) to asynchronously load the data, thereby further reducing load time.

```r
# this would placed in the shiny UI
load_json_from_ui <- function(path_to_json){
  script <- paste0("
    $.ajax({
        url: '", path_to_json, "',
        dataType: 'json',
        async: true,
        success: function(data){
          console.log(data);
          window.globalData = data;
        }
      });"
    )
  shiny::tags$script(
    script
  )
}
```

Using the above the data loaded would be accessible from the htmlwidgets JavaScript (e.g.: `gio.js`) with `window.globalData`. The `window` object is akin to the `document` object, while the latter pertains to the Document Object Model (DOM) and represents the page, the former pertains to the Browser Object Model (BOM) and represents the browser window. While `var x;` will only be accessible within the script where it is declared, `window.x` will be accessible anywhere.

Note this means the data is read from the web browser and therefore the data must be accessible to the web browser, the `path_to_json` must thus be a served static file, e.g.: `www` directory in shiny.

9

Crosstalk

Crosstalk (**?**) is fantastic add-on for htmlwidgets that implements cross-widget interactions, namely selection and filtering. This in effect allows the selection or filtering of data points in one widget to be mirrored in another. This is enabled by the creation of "shared datasets" that can be used across widgets.

Crosstalk provides a straightforward interface to the users and instead requires effort from the developers for their widgets to support the shared datasets.

9.1 Crosstalk example

Both the plotly and DT packages support crosstalk, we can thus produce a scatter plot with the former and a table with the latter so that selection of data in one widget is reflected in the other.

The shared dataset is created with the `SharedData` R6 class, this dataset is then used as one would use a standard dataframe in plotly and DT. The `bscols` function is just a helper to create columns from html elements (using bootstrap). It's ideal for examples but one should not have to use it in Shiny—crosstalk will work without `bscols`.

```
library(DT)
library(plotly)
library(crosstalk)

shared <- SharedData$new(cars)

bscols(
  plot_ly(shared, x = ~speed, y=~dist),
  datatable(shared, width = "100%")
)
```

With crosstalk comes the concept of "groups," it defines an instance of shared