

Join us on **Nov 20, 2024, in Basel** for the **GxP Validation Summit: Clinical Software Development**. [Register Now](#)

[Services](#) ▾[Case
Studies](#)[Data4Good](#)[Careers](#)[Resources](#)[Blog](#)[Talk to our
Experts](#)

Webscraping Dynamic Websites with R

Estimated time: 15 min

Date: September 1, 2022



author: **Ivan Millanes**

[r](#)[tutorial](#)

In this post, you'll learn how to scrape dynamic websites in R using {RSelenium} and {rvest}. Although some basic knowledge of rvest, HTML, and CSS is required, I will explain basic concepts through the post. So even beginners will find some use in this tutorial for webscraping dynamic sites in R.

You can do a lot with R these days. Discover these 6 essential R packages from [scraping webpages to training ML models](#).

TOC:

- [Static vs Dynamic Web Pages](#)
- [Setup](#)
- [Basics of Webscraping in R](#)

**Have
questions
or
insights?**

Engage with experts, share ideas and take your data journey to the next level!

[Contact
Us](#)

[Join Slack](#)

Your data journey has begun.
Equip yourself with the right tools.

- [Example of Webscraping in R](#)
 - [Tips for Working with RSelenium](#)
-

Static vs Dynamic Web Pages

Let's compare the following websites:

- [IMDB](#) - an internet movie database
- [Premier League](#) - a site containing football (soccer) statistics and info

On IMDB, if you search for a particular movie (e.g. [The Dark Knight](#)), you can see that the URL changes and the URL is different from any other movie (e.g. [Predestination](#)).

On the other hand, if you go to [Premier League Player Stats](#), you'll notice that modifying the filters or clicking the pagination button to access more data doesn't produce changes to the URL.

The first website is an example of a **static** web page, whereas the second is an example of a **dynamic** web page.

- **Static Web Page:** A web page (HTML page) that contains the same information for all users. Although it may be periodically updated, it does not change with each user retrieval.
- **Dynamic Web Page:** A web page that provides custom content for the user based on the results of a search or some other request. Also known as "dynamic HTML" or "dynamic content", the "dynamic" term is used when referring to interactive Web pages created for each user.

If you're looking to scrape data from static web pages - '[rvest](#)' is a great tool.

Thinking about a career in R and R Shiny? [Here's everything you need to know to land your first R Developer job.](#)

But when it comes to dynamic web pages, `rvest` alone can't get the job done. This is when [RSelenium](#) joins the party.

Setup for Webscraping Dynamic Websites in R

R Packages

Before we dive into the details and how to get started , make sure you install the following packages and load them so you can run the code written below:

```
```{r, message=FALSE, warning=FALSE}
library(dplyr)
library(stringr)
library(purrr)
library(rvest)
library(RSelenium)
```
```

Java

It's also important that you have Java installed. To check the installation, type `java -version` in your Command Prompt. If it throws an error, it means you don't have Java installed.

You can [download Java here](#).

Start Selenium

Use [rsDriver\(\)](#) to start a Selenium server and browser. If not specified, `browser = "chrome"` and `version = "latest"` are the default values for those parameters.

```
```{r, eval=FALSE}
rD <- RSelenium::rsDriver() # This might throw an error
```
```

The code above might throw an error that looks like this:

```
> rD <- RSelenium::rsDriver()
checking Selenium Server versions:
BEGIN: PREDOWNLOAD
BEGIN: DOWNLOAD
BEGIN: POSTDOWNLOAD
checking chromedriver versions:
BEGIN: PREDOWNLOAD
BEGIN: DOWNLOAD
BEGIN: POSTDOWNLOAD
checking geckodriver versions:
BEGIN: PREDOWNLOAD
BEGIN: DOWNLOAD
BEGIN: POSTDOWNLOAD
checking phantomjs versions:
BEGIN: PREDOWNLOAD
BEGIN: DOWNLOAD
BEGIN: POSTDOWNLOAD
[1] "Connecting to remote server"

selenium message:session not created: This version of ChromeDriver only supports Chrome version 104
Current browser version is 103.0.5060.66 with binary path C:\Program Files (x86)\Google\Chrome\Application\chrome.exe
Build info: version: '4.0.0-alpha-2', revision: 'f148142cf8', time: '2019-07-01T21:30:10'
System info: host: 'LAPTOP-KLALVIRJ', ip: '192.168.0.93', os.name: 'windows 10', os.arch: 'amd64', os.version: '10.0',
java.version: '1.8.0_333'
Driver info: driver.version: unknown
```

You can explore [this StackOverflow post](#) explaining what the error is about. Basically, there is a mismatch between the Chrome driver and the Chrome browser versions. [This solution](#) is to set 'chromever' parameter to the latest compatible Chrome driver version.

I'll show you how to determine a proper value by manually checking versions. First, we need to identify what Chrome version we have. You can do that with the following code:

```
```{r eval=FALSE}
Get Chrome version
system2(command = "wmic",
 args = 'datafile where name="C:\\\\Program Files (x86)\\\\Goog
```
```

If you run that code in the console, you should see a result that looks like this (note: your version may differ from the one shown here):

```
> system2(command = "wmic",
+         args = 'datafile where name="C:\\\\Program Files (x86)\\\\Google\\\\Chrome\\\\Application\\\\chrome.exe" get Version /value')
Version=103.0.5060.66
```

Now we have to list the available Chrome drivers:

```
` `{r eval=FALSE}
binman::list_versions(appname = "chromedriver")
` `
```

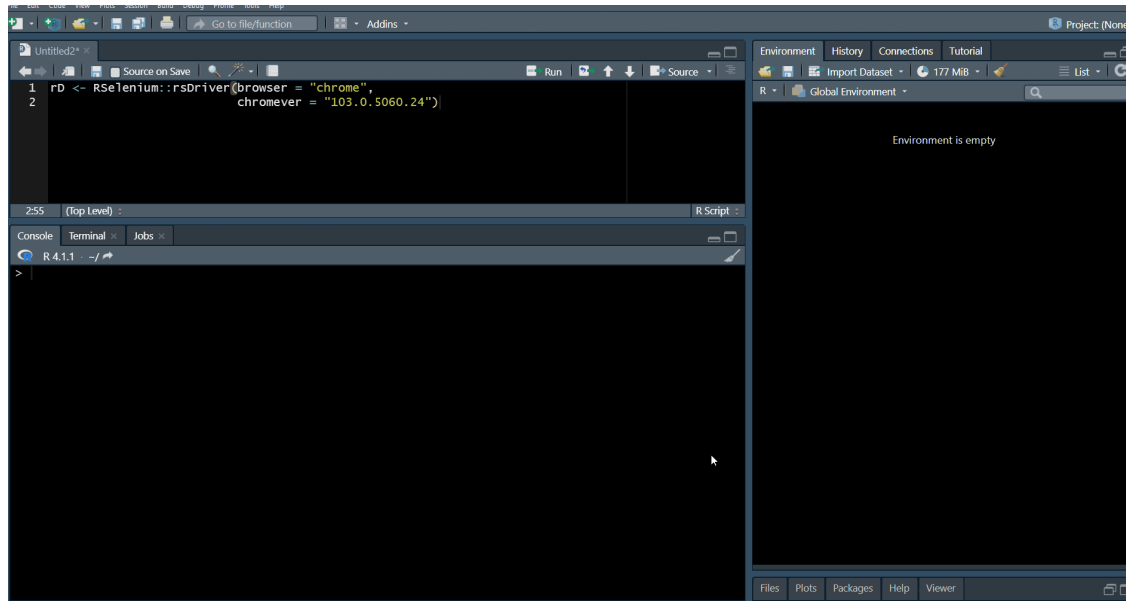
```
> binman::list_versions(appname = "chromedriver")
$win32
[1] "103.0.5060.24" "103.0.5060.53" "104.0.5112.20" "81.0.4044.138"
[5] "83.0.4103.39"  "84.0.4147.30"  "85.0.4183.38"  "88.0.4324.27"
[9] "88.0.4324.96"  "89.0.4389.23"  "97.0.4692.36"  "97.0.4692.71"
[13] "98.0.4758.48"
```

Each version of the Chrome driver supports Chrome with matching major, minor, and build version numbers. For example, Chrome driver `73.0.3683.20` supports all Chrome versions that start with `73.0.3683`. In our case, we could use either `103.0.5060.24` or `103.0.5060.53`. In the case that there is no Chrome driver matching the Chrome version you'll need to install it.

The updated code looks like this:

```
` `{r, eval=FALSE}
# Start Selenium server and browser
rD <- RSelenium::rsDriver(browser = "chrome",
                           chromeversion = "103.0.5060.24")
# Assign the client to an object
remDr <- rD[["client"]]
` `
```

Running `rD <- RSelenium::rsDriver(...)` should open a new Chrome window.



You can find more information about `rsDriver()` in the [Basics of Vignettes article](#).

Basics of R and Webscraping

In this section, I'll apply different methods to the `remDr` object created above. I'm only going to describe the methods that I think are most frequently used. For a complete reference, check the [package documentation](#).

- `navigate(URL)` : Navigate to a given URL

```
```{r, eval=FALSE}
remDr$navigate("https://www.google.com/")
remDr$navigate("https://www.nytimes.com/")

Use method without () to get a description of what it does
remDr$navigate
```
```

- `goBack()` : Equivalent to hitting the back button on the browser
- `goForward()` : Equivalent to hitting the forward button on the browser

```
```{r, eval=FALSE}
remDr$goBack()
remDr$goForward()
```
```

- **refresh()** : Reload the current page

```
```{r, eval=FALSE}
remDr$refresh()
```
```

- **getCurrentUrl()** : Retrieve the URL of the current page

```
```{r, eval=FALSE}
remDr$getCurrentUrl()
```
```

- **maxWindowSize()** : Set the size of the browser window to maximum. By default, the browser window size is small, and some elements of the website you navigate to might not be available right away (I'll talk more about this in the next section).

```
```{r, eval=FALSE}
remDr$maxWindowSize()
```
```

```
```
```

- `getPageSource() [[1]]` : Get the current page source. This method combined with ``rvest`` is what makes possible to scrape dynamic web pages. The xml document returned by the method can then be read using `rvest::read_html()`. This method returns a ``list`` object, that's the reason behind `[[1]]`.

```
```{r, eval=FALSE}
remDr$getPageSource() [[1]]
```
```

- `open(silent = FALSE)` : Send a request to the remote server to instantiate the browser. I use this method when the browser closes for some reason (for example, inactivity). If you have already started the Selenium server, you should run this instead of `rD <- RSelenium::rsDriver(...)` to re-open the browser.

```
```{r, eval=FALSE}
remDr$open()
```
```

- `close()` : Close the current session

```
```{r, eval=FALSE}
remDr$close()
```
```



## Working with Elements

- `findElement(using, value)` : Search for an element on the page, starting from the document root. The located element will be returned as an object of `webElement` class. To use this function you need some basic knowledge of HTML and CSS (or xpath, etc). Using a Chrome extension, called [SelectorGadget](#), might help.
- `highlightElement()` : Utility function to highlight current Element. This helps to check that you selected the wanted element.
- `sendKeysToElement()` : Send a sequence of keystrokes to an element. The keystrokes are sent as a list. Plain text is entered as an unnamed element of the list. Keyboard entries are defined in 'selKeys' and should be listed with the name 'key'.
- `clearElement()` : Clear a TEXTAREA or text INPUT element's value.
- `clickElement()` : Click the element. You can click links, check boxes, dropdown lists, etc.

## Example of Working with Elements in R

To understand the following example, basic knowledge of CSS is required.

```
```{r, eval=FALSE}
# Navigate to Google
remDr$navigate("https://www.google.com/")

# Find search box
webElem <- remDr$findElement(using = "css selector", value = ".gLfyf.g

# Highlight to check that was correctly selected
webElem$highlightElement()

# Send search and press enter
# Option 1
webElem$sendKeysToElement(list("the new york times"))
webElem$sendKeysToElement(list(key = "enter"))
# Option 2
```

```

webElem$sendKeysToElement(list("the new york times", key = "enter"))

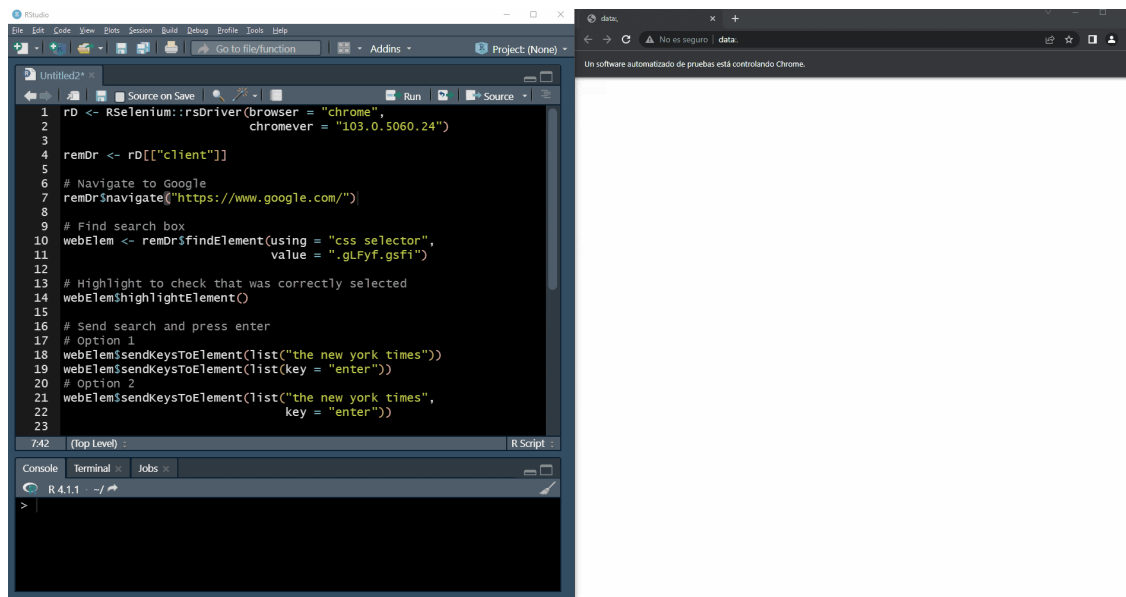
# Go back to Google
remDr$goBack()

# Search something else
webElem$sendKeysToElement(list("finantial times"))

# Clear element
webElem$clearElement()

# Search and click
webElem <- remDr$findElement(using = "css selector", value = ".gLfyf.g
webElem$sendKeysToElement(list("the new york times", key = "enter"))
webElem <- remDr$findElement(using = "css selector", value = ".LC20lb.
webElem$clickElement()
` ``

```



Other Methods of Webscraping with R

In this section, I'll list other methods that might be useful to you. For more information about each, be sure to explore the [RSelenium documentation](https://seleniumhq.github.io/selenium/docs/api/javascript/).

```

` ``{r, eval=FALSE}
remDr$status()

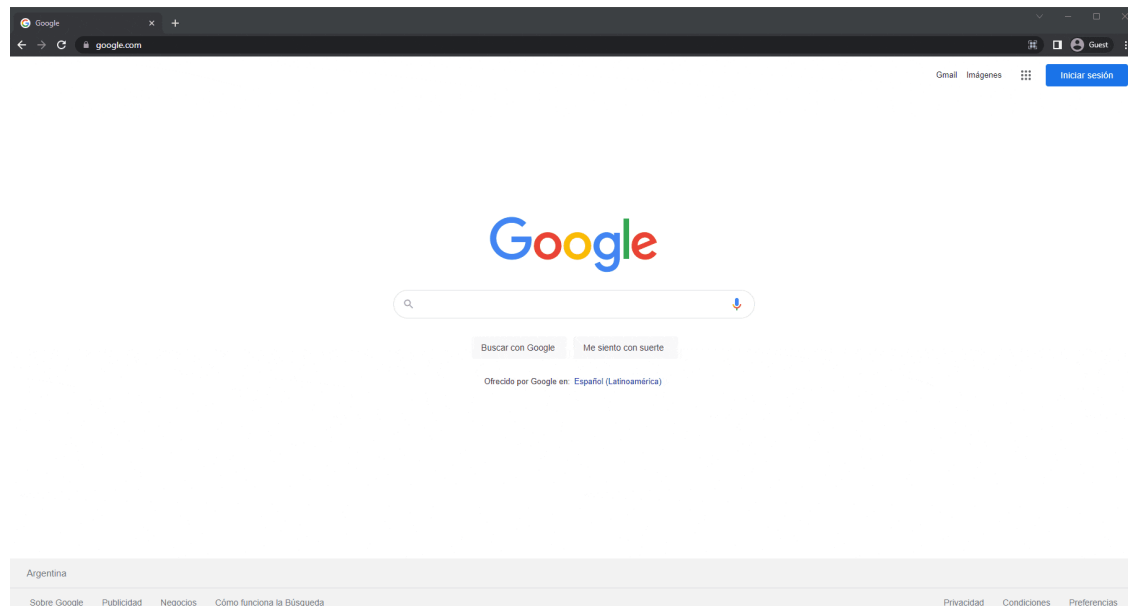
```

```
remDr$title()
remDr$screenshot()
remDr$getWindow()
remDr$setWindowSize(1000,800)
remDr$getWindowPosition()
remDr$setWindowPosition(100, 100)
webElem$getElementLocation()
````
```

## Example of Webscraping Premier League Player Goals with R

In this example, I'll create a dataset using information stored on the Premier League Player stats page, which we discussed earlier.

First, let's explore the site.



There are a couple of interesting things to point out:

- When we open the website, we are asked to accept cookies.
- After we accept cookies, an ad opens which has to be closed.
- As expected, selecting a different season and table pagination produce no changes in the URL.

In our code, we'll have to include commands to navigate to the website, accept cookies, and close the ad. Note that the website might change in the future, so with time some modifications to the following code might be necessary.

## Target Dataset

Our final dataset will contain the following variables:

- **Player** : Indicates the player's name.
- **Nationality** : Indicates the nationality of the player.
- **Season** : Indicates the season the stats corresponds to.
- **Position** : Indicates the player's position in the season.
- **Goals** : Number of Goals scored by the player.

For simplicity's sake, we'll scrape data from seasons 2017/18 and 2018/19.

## Before We Start

In order to run the code below, we have to start a Selenium server and browser. And we'll need to create the `remDr` object. This step was described in the [Start Selenium](#) section.

```
` `{r, eval=FALSE}
Start Selenium server and browser
rD <- RSelenium::rsDriver(browser = "chrome",
 chromeversion = "103.0.5060.24") # You might hav
Assign the client to an object
remDr <- rD[["client"]]
` `
```

## First Steps

The code chunk below:

- Navigates to the website
- Increases the window size (This action might be useful to show elements that might be hidden due to window size.)
- Accepts cookies
- Closes the add

You might notice two things:

- The use of the `Sys.sleep()` function.

This function is used to give the website enough time to load. Sometimes, if the element you want to find isn't loaded when you search for it, it will produce an error.

- The use of CSS selectors.

To select an element using CSS you can press F12 and inspect the page source (right-clicking the element and selecting **Inspect** will show you which part of that code refers to the element). Or you can use a chrome extension, called [SelectorGadget](#). I recommend learning some HTML and CSS and using these two approaches simultaneously. SelectorGadget helps, but sometimes you will need to inspect the source to get exactly what you want.

```
```{r, eval=FALSE}
# Navigate to the website
remDr$navigate("https://www.premierleague.com/stats/top/players/goals")
# Give some time to load
Sys.sleep(4)
# Increase window size to find elements
remDr$maxWindowSize()
# Accept cookies
acceptCookies <- remDr$findElement(using = "css selector",
                                     value = ".js-accept-all-close")
acceptCookies$clickElement()
# Give some time to load
Sys.sleep(2)
```

```
# Close add
closeAdd <- remDr$findElement(using = "css_selector",
                               value = "#advertClose")
closeAdd$clickElement()
````
```

In the next subsection, I'll show how I selected certain elements by inspecting the page source.

## Getting values to Iterate Over

In order to get the data, we will have to iterate over different lists of values. In particular, we need a list of seasons and player positions.

We can use `rvest` to scrape the website and get these lists. To do so, we need to find the corresponding nodes. As an example, after the code, I'll show where I searched for the required information in the page source for the seasons' lists.

The code below uses `rvest` to create the lists we'll use in the loops.

```
````{r, eval=FALSE}
# Read page source
source <- remDr$getPageSource()[[1]]
# Get seasons
list_seasons <- read_html(source) %>%
  html_nodes("ul[data-dropdown-list=FOOTBALL_COMPSEASON] > li") %>%
  html_attr("data-option-name") %>%
  .[-1] # Remove "All seasons" option
# To make example simple
season17 <- which(list_seasons == "2017/18")
season18 <- which(list_seasons == "2018/19")
list_seasons <- list_seasons[c(season17, season18)]
# Get positions
list_positions <- read_html(source) %>%
  html_nodes("ul[data-dropdown-list=Position] > li") %>%
  html_attr("data-option-id") %>%
  .[-1] # Remove "All positions" option
````
```

## Seasons

This is my view when I open the seasons dropdown list and right-click and inspect the 2016/17 season:

The screenshot shows the Premier League Player Stats page. The 'Stats Centre' sidebar is on the left. The main content area is titled 'Premier League Player Stats'. A dropdown menu for 'Filter by Season' is open, showing a list of seasons from 2019/20 to 2015/16. The 2016/17 season is highlighted. The right sidebar shows the browser's developer tools with the 'Elements' panel open, displaying the HTML structure of the dropdown menu. The HTML includes a `<ul class="dropdownList" data-dropdown-list="FOOTBALL_COMPSEASON" role="listbox" aria-labelledby="dd-FOOTBALL_COMPSEASON" data-listen-keypress="true" data-listen-click="true">` tag, followed by a `::before` pseudo-class and a list of `<li>` elements, each with a `role="option"` and a `data-option-name` attribute. The 2016/17 season is highlighted in the list.

Taking a closer look at the source where that element is present we get:

```
<ul class="dropdownList" data-dropdown-list="FOOTBALL_COMPSEASON"
role="listbox" aria-labelledby="dd-FOOTBALL_COMPSEASON" data-listen-
keypress="true" data-listen-click="true">
 ::before
 <li role="option" tabindex="0" data-option-name="All Seasons"
data-option-id="-1" data-option-index="-1">All Seasons
 <li role="option" tabindex="0" data-option-name="2019/20" data-
option-id="274" data-option-index="0">2019/20
 <li role="option" tabindex="0" data-option-name="2018/19" data-
option-id="210" data-option-index="1">2018/19
 <li role="option" tabindex="0" data-option-name="2017/18" data-
option-id="79" data-option-index="2">2017/18
 <li role="option" tabindex="0" data-option-name="2016/17" data-
option-id="54" data-option-index="3">2016/17 == $0
 <li role="option" tabindex="0" data-option-name="2015/16" data-
```

As you can see, we have an attribute named 'data-dropdown-list' whose value is 'FOOTBALL\_COMPSEASON' and inside we have 'li' tags where the attribute 'data-option-name' changes for each season. This will be useful when defining how to iterate using 'RSelenium'.

## Web scraping Loop in R

This is an overview of the loop to get `Goals` data.

- Preallocate seasons vector. This list will have a length equal to the number of seasons to be scraped.
- For each season:
  - Click the seasons dropdown list
  - Click the corresponding season
  - Preallocate positions vector. This list will have `length = 4` (positions are fixed: GOALKEEPER, DEFENDER, MIDFIELDER, and FORWARD).
  - For each position inside the season
    - Click the position dropdown list
    - Click the corresponding position
    - Check that there is a table with data (if not, go to next position)
    - Scrape the first table
    - While "Next Page" button exists
      - Click "Next Page" button
      - Scrape new table
      - Append new table to table
- Go to the top of the website
- Rowbind each position table
- Add season data



- Rowbind each season table to create `Goals` datasetThe result of this loop is a `tibble`.

This is the code:

```
``{r, eval=FALSE}
Preallocate seasons vector
data_seasons <- vector("list", length(list_seasons))
Note: DDL is short for DropDown List
Iterate over seasons
for (j in seq_along(list_seasons)){

 # Open seasons dropdown list
 DDLseason <- remDr$findElement(using = "css selector",
 value = ".current[data-dropdown-curre

 DDLseason$clickElement()
 Sys.sleep(2)

 # Click corresponding season
 ELEMseason <- remDr$findElement(using = "css selector", value = str_
 ELEMseason$clickElement()
 Sys.sleep(2)

 # Preallocate positions vector
 data_positions <- vector("list", length(list_positions))

 # Iterate over position
 for (k in seq_along(list_positions)){
 # Open positions dropdown list
 DDLposition <- remDr$findElement(using = "css selector",
 value = ".current[data-dropdown-c

 DDLposition$clickElement()
 Sys.sleep(2)

 # Click corresponding position
 ELEMposition <- remDr$findElement(using = "css selector", value =
 ELEMposition$clickElement()
 Sys.sleep(2)
```

```

Check that there is a table to scrape. If there isn't, go to next
check_table <- remDr$getPageSource()[[1]] %>%
 read_html() %>%
 html_node(".statsTableContainer") %>%
 html_text()

if(check_table == "No stats are available for your search") next

Populate element of corresponding position (first page)
data_positions[[k]] <- remDr$getPageSource()[[1]] %>%
 read_html() %>%
 html_table() %>%
 .[[1]] %>%
 # Process was including a column without name which we need to r
 # To do so, we include the following lines of code.
 as_tibble(.name_repair = "unique") %>%
 select(-ncol(.))

Get tables from every page
btnNextExists <- remDr$getPageSource()[[1]] %>%
 read_html() %>%
 html_node(".paginationNextContainer.inactive") %>%
 html_text() %>%
 is.na()

While there is a Next button to click
while (btnNextExists){
 # Click "Next"
 btnNext <- remDr$findElement(using = "css selector",
 value = ".paginationNextContainer")

 btnNext$clickElement()
 Sys.sleep(2)

 # Get table from new page
 table_n <- remDr$getPageSource()[[1]] %>%
 read_html() %>%
 html_table() %>%
 .[[1]] %>%
 # Process was including a column without name which we need to
 # To do so, we include the following lines of code.
 as_tibble(.name_repair = "unique") %>%
 select(-ncol(.))

 # Rowbind existing table and new table
 data_positions[[k]] <- bind_rows(data_positions[[k]], table_n)

 # Update Check for Next Button
 btnNextExists <- remDr$getPageSource()[[1]] %>%
 read_html() %>%
 html_node(".paginationNextContainer.inactive") %>%
 html_text() %>%

```

```

 is.na()

 Sys.sleep(1)
 }

 # Data wrangling
 data_positions[[k]] <- data_positions[[k]] %>%
 rename(Goals = Stat) %>%
 mutate(Position = list_positions[[k]])

 # Go to top of the page to select next position
 goTop <- remDr$findElement("css", "body")
 goTop$sendKeysToElement(list(key = "home"))
 Sys.sleep(3)
}

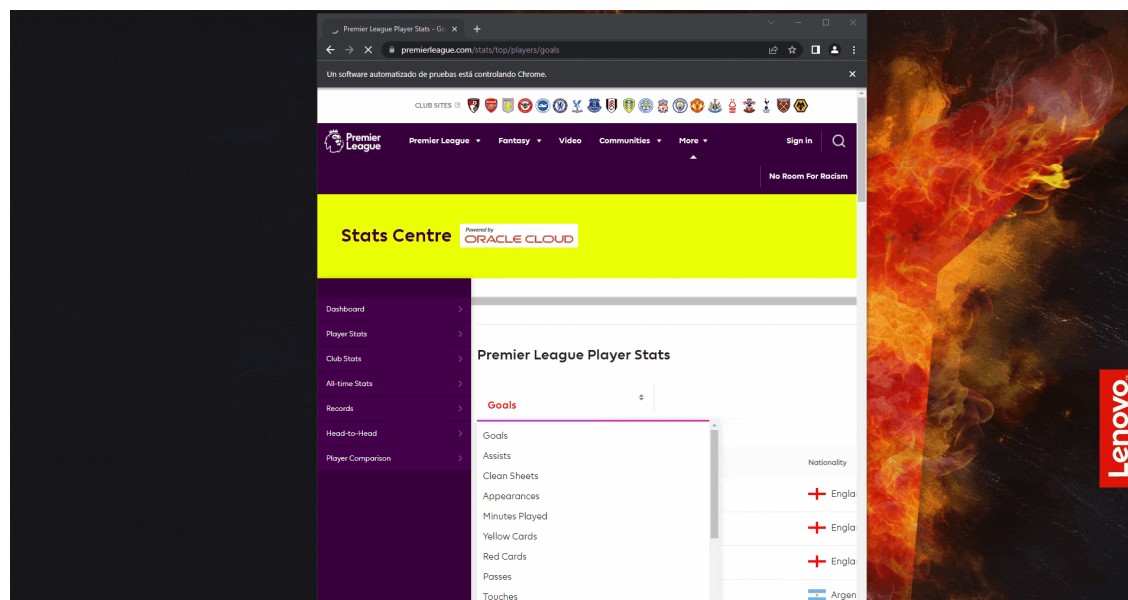
Rowbind positions dataset
data_positions <- reduce(data_positions, bind_rows)

Populate corresponding season
data_seasons[[j]] <- data_positions %>%
 mutate(Season = list_seasons[[j]])
}

Rowbind seasons dataset to create goals dataset
data_goals <- reduce(data_seasons, bind_rows)
```

```

This is how the scraping looks in action:



Final Dataset

After some data wrangling, this is how the final dataset looks:

```
```{r, eval=FALSE}
dataset <- data_goals %>%
 select(-c(Rank, Club)) %>%
 select(Season, Position, Player, Nationality, Goals) %>%
 arrange(Season, Position, Player, Nationality)
```

```{r, echo=FALSE}
dataset <- readRDS("data/dataset.rds") ```{r} dataset %>%
 head %>%
 knitr::kable(format = "html")
```
```

Tips for Working with RSelenium for Webscraping

In this section, I'll discuss general topics that might help you when working with `RSelenium`. I'll also cover how to troubleshoot some issues that I've experienced in the past.

Parallel Framework

The framework described here is an approach to working in `parallel` with `RSelenium`. This way you can open multiple browsers at the same time and speed up the scraping. Be careful though, because I have experienced issues such as browsers closing for no apparent reason while working in parallel.

First, we load the libraries we need:

```
```{r, eval=FALSE}
Load libraries
```

```
library(parallel)
````
```

We then determine the number of cores to use. In this example, I use four cores.

```
````{r, eval=FALSE}
Determine cores
Number of cores in your computer
n_cores <- detectCores()

It's recommended to always leave at least one core free
clust <- makeCluster(n_cores - 1)

I decided to make an example using 4 cores.
clust <- makeCluster(4)
````
```

List the ports that are going to be used to start selenium:

```
````{r, eval=FALSE}
List ports
ports = list(4567L, 4444L, 4445L, 5555L)
````
```

We use the `clusterApply()` to start Selenium on each core. Pay attention to the use of the [superassignment operator](#). When you run this function, you will see that four chrome windows are opened.

```
````{r, eval=FALSE}
Open Selenium on each core, using one port per core.
clusterApply(clust, ports, function(x){
 # Here you load the libraries on each core
 library(RSelenium)
 library(dplyr) # Not needed for this particular example
})
```

```

library(rvest) # Not needed for this particular example

Pay attention to the use of the superassignment operator.
rD <-> RSelenium::rsDriver(
 browser = "chrome",
 chromeversion = "103.0.5060.24",
 port = x
)

Pay attention to the use of the superassignment operator.
remDr <-> rD[["client"]]
})
```

```

This is an example of pages that we will open in parallel:

```

```{r, eval=FALSE}
List element to iterate with parallel processing
pgs <- list("https://www.google.com",
 "https://www.nytimes.com",
 "https://www.ft.com")
```

```

Use 'parLapply()' to work in parallel. When you run this, you'll see that each browser opens one website, and one remains blank. This is a simple example; I haven't defined any scraping, but we can!

```

```{r, eval=FALSE}
Define iteration
parLapply(clust, pgs, function(x) {
 remDr$navigate(x)
})
```

```

When you're done, stop Selenium on each core and stop the cluster.

```

```{r, eval=FALSE}
Define function to stop Selenium on each core
close_rselenium <- function(){
 clusterEvalQ(clust, {
 remDr$close()
 rD$server$stop()
 })
 system("taskkill /im java.exe /f", intern=FALSE, ignore.stdout=FALSE)
}
```

```{r, eval=FALSE}
Close Selenium on each core
close_rselenium()
Stop the cluster
stopCluster(clust)
```

```

Browser Closing for No Reason

Consider the following scenario: your loop navigates to a certain website, clicks some elements, and then gets the page source to scrape using ``rvest``. If in the middle of that loop the browser closes, you will get an error (e.g., it won't navigate to the website or the element won't be found).

You can work around these errors using ``tryCatch()``, but when you skip the iteration where the error occurred when you try to navigate to the website in the following iteration, an error will occur again (because there is no browser open!).

You could, for example, use `remDr$open()` at the beginning of the loop, and ``remDr$close()`` at the end, but that may open and close too many browsers and make the process slower.

So I created this function that handles part of the problem. Even though the iteration where the browser is closed will not finish, the next one will. And the process won't stop.

It basically tries to get the current URL using `remDr$getCurrentUrl()`. If no browser is open, this will throw an error, and if we get an error, it will open a browser.

```

```{r, eval=FALSE}
check_chrome <- function(){
 check <- try(suppressMessages(remDr$getCurrentUrl()), silent = TRUE)
 if ("try-error" %in% class(check)) remDr$open(silent = TRUE)
}
```

```

Closing Selenium (Port Already in Use)

Sometimes, even if the browser window is closed when you re-run `rD <-`

`RSelenium::rsDriver(...)` you might encounter an error like:

```

```
Error in wdman::selenium(port = port, verbose = verbose, version = ver
 Selenium server signals port = 4567 is alrea
```

```

This means the connection was not completely closed. You can execute the lines of code below to stop

Selenium:

```

```{r, eval=FALSE}
remDr$close()
rD$server$stop()
system("taskkill /im java.exe /f", intern=FALSE, ignore.stdout=FALSE)
```

```

You can check out [this Stackoverflow post](#) for more info.

Wrapper Functions

You can create functions in order to type less. Suppose that you navigate to a certain website where you have to click one link that sends you to a site with different tabs. You can use something like this:

```
```{r}
navigate_page <- function(CSS_ID, CSS_TAB = NULL){
 remDr$navigate("WEBSITE")
 webElem <- remDr$findElement(using = "css selector", CSS_ID)
 webElem$clickElement()
 if (!is.null(TAB)){
 tab <- remDr$findElement(using = "css selector", CSS_TAB)
 tab$clickElement()
 }
}
}```
```

You can also create functions to find elements, check if an element exists on the [DOM \(Document Object Model\)](#), try to click an element if it exists, parse the data table you are interested in, etc. You might find these [StackOverflow examples](#) helpful.

Looking to create interactive Markdown documents? Explore [R Quarto with our tutorial to get you started](#).

## Resources for RSelenium Projects

The following list contains different videos, posts, and StackOverflow posts that I found useful when learning and working with RSelenium.

- The ultimate online collection toolbox: Combining RSelenium and Rvest [\[Part I\]](#) and [\[Part II\]](#).

If you know about `rvest` and just want to learn about `RSelenium`, I'd recommend watching Part II. It gives an overview of what you can do when combining `RSelenium` and `rvest`. It has decent, practical examples. As a final comment regarding these videos, I wouldn't pay too much attention to setting up Docker because you don't need to work that way in order to get `RSelenium` going.

- R Selenium Tutorial: A Tutorial to Basic Web Scraping With R Selenium [\[Link\]](#).

I found this post really useful when trying to set up `R Selenium`. The solution given in [this StackOverflow post](#), which is mentioned in the article, seems to be enough.

- Dungeons and Dragons Web Scraping with rvest and R Selenium [\[Link\]](#).

This is a great post! It starts with a general tutorial for scraping with `rvest` and then dives into `R Selenium`. If you are not familiar with `rvest`, you should start here.

- R Selenium Tutorial [\[Link\]](#).
- R Selenium Package Website [\[Link\]](#).

It has more advanced and detailed content. I just took a look at the [Basics vignette](#).

These StackOverflow posts helped me when working with dropdown lists:

- R selenium - How to scrape all drop-down list option values [\[Post\]](#)
- Dropdown boxes in R Selenium [\[Post\]](#)

This post gives a solution to the "port already in use" problem:

- R Selenium: server signals port is already in use [\[Post\]](#).

Even though is not marked as "best," the last line of code of the second answer is useful.

### Enhancing Shiny Application Performance: Implementing Efficient Coding Practices

Learn how to optimize your Shiny application's performance with best coding practices.



Jakub Chojna

### A Guide to R Package Validation in Pharma

Learn how to validate R packages for GxP compliance in pharma. Ensure accuracy, reproducibility, and regulatory adherence with our step-by-step guide.



Gift Kenneth

### Rhino 1.10.0 Update: Automated Styling & Auto-complete for box Modules

Rhino 1.10.0 brings easier coding with automated styling and auto-complete for box modules—keeping your code clean and efficient.




Ricardo Rodrigo Basa

[Explore Possibilities](#)

## Share Your Data Goals with Us

From advanced analytics to platform development and pharma consulting, we craft solutions tailored to your needs.

[Talk to our Experts](#)



Appsilon

Join 4000+ Shiny enthusiasts to see the latest Shiny news from the R community.

Name

email

Submit

☐ By completing the form, you agree to receive commercial information by email from Appsilon. You can withdraw consent at any time.

Company

Data4Good

AI and Research

Blog

Careers

Privacy Policy

Code of Conduct

Shiny Resources

Shiny Conference

Shiny Weekly

Shiny Templates

Shiny Tools

Appsilon's GitHub

Partnerships

Posit (formerly RStudio)

Domino Data Lab

R-Bloggers

Python-Bloggers