**Single web-page (Wikipedia)**

library(rvest)

library(polite)

library(dplyr)

library(tidyr)

library(purrr)

library(stringr)

library(glue)

library(rlang)

For the first example, let’s start with scraping soccer data from  
Wikipedia, specifically the top goal scorers of the Asian Cup.

We use polite::bow() to pass the URL for the Wikipedia article to get  
a polite session object. This object will tell you about the  
robots.txt, the recommended crawl delay between scraping attempts, and  
tells you whether you are allowed to scrape this URL or not. You can  
also add your own user name in the user\_agent argument to introduce  
yourself to the website.

topg\_url <- "<https://en.wikipedia.org/wiki/AFC_Asian_Cup_records_and_statistics>"

session <- bow(topg\_url,

user\_agent = "Ryo's R Webscraping Tutorial")

session

## <https://en.wikipedia.org/wiki/AFC_Asian_Cup_records_and_statistics>

## User-agent: Ryo's R Webscraping Tutorial

## robots.txt: 454 rules are defined for 33 bots

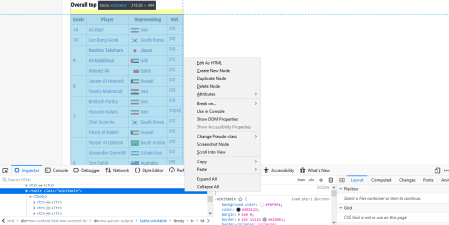
## Crawl delay: 5 sec

## The path is scrapable for this user-agent

Now to actually get the data from the webpage. You’ve got different  
options depending on what browser you’re using but on Google Chrome or  
Mozilla Firefox you can find the exact HTML element by right clicking on  
it and then clicking on “Inspect” or “Inspect Element” in the pop-up  
menu. By doing so, a new view will open up showing you the full HTML  
content of the webpage with the element you chose highlighted. (See first two pics)

You might also want to try using a handy JavaScript tool called SelectorGadget,  
you can learn how to use it [here](https://rvest.tidyverse.org/articles/selectorgadget.html). It allows you to click on different elements of the web page and  
the gadget will try to ascertain the exact CSS Selector in the HTML. (See bottom pic)

Do be warned that web pages can change suddenly and the CSS Selector you  
used in the past might not work anymore. I’ve had this happen more than a few times  
as pages get updated with more info from new tournaments and such. This  
is why you really should try to scrape from a more stable website, but a  
lot of times for “simple” data Wikipedia is the easiest and best place  
to scrape.



From here you can right-click again on the highlighted HTML code to  
“Copy”, and then you can choose one of “CSS Selector”, “CSS Path”, or  
“XPath”. I normally use “CSS Selector” and it will be the one I will use  
throughout this tutorial. This is the exact reference within the HTML  
code of the webpage of the object you want. I make sure to choose the  
CSS Selector for the **table** itself and not just the info inside the  
table.

With this copied, you can go to your R script/RMD/etc. After running the  
polite::scrape() function on your bow object, paste in the CSS  
Selector/Path/XPath you just copied into html\_nodes(). The bow  
object already has the recommended scrape delay as stipulated in a  
website’s robots.txt so you don’t have to input it manually when you  
scrape.

ac\_top\_scorers\_node <- scrape(session) %>%

html\_nodes("table.wikitable:nth-child(44)")

Grabbing a HTML table is the easiest way to get data as you usually  
don’t have to do too much work to reshape the data afterwards. We can do  
that with the html\_table() function. As the HTML object returns as a  
list, we have to flatten it out one level using purrr::flatten\_df() .  
Finish cleaning it up by taking out the unnecessary “Ref” column with  
select() and renaming the column names with set\_names().

ac\_top\_scorers <- ac\_top\_scorers\_node %>%

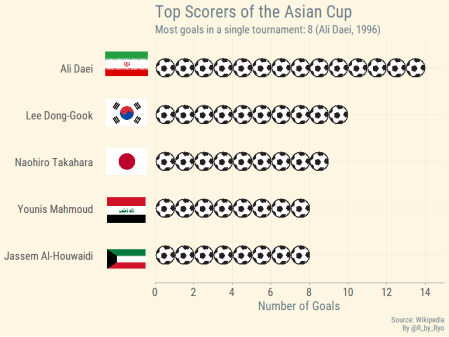
html\_table() %>%

flatten\_df() %>%

select(-Ref.) %>%

set\_names(c("total\_goals", "player", "country"))

After adding some flag and soccer ball images to the data.frame we get  
this:



Do note that the image itself is from **before** the 2019 Asian Cup but  
the data we scraped in the code above is **updated**. As a visualization  
challenge try to create a similar viz with the updated data!

**Single-page (Transfermarkt)**

So now let’s try a soccer-specific website as that’s really the goal of  
this blog post. This time we’ll go for one of the most famous soccer  
websites around, [transfermarkt.com](http://transfermarkt.com). A website used as a data source  
from your humble footy blogger to big news sites such as the [Financial  
Times](https://twitter.com/jburnmurdoch/status/1058036551454724096) and  
the [BBC](https://www.bbc.com/sport/football/52221463).

The example we’ll try is from an Age-Value graph for the J-League I made  
around 2 years ago when I just started doing soccer data viz (how times  
flies…).

url <- "<https://www.transfermarkt.com/j-league-division-1/startseite/wettbewerb/JAP1/saison_id/2017>"

session <- bow(url)

session

## <https://www.transfermarkt.com/j-league-division-1/startseite/wettbewerb/JAP1/saison_id/2017>

## User-agent: polite R package -

## robots.txt: 1 rules are defined for 1 bots

## Crawl delay: 5 sec

## The path is scrapable for this user-agent

* [Terms of Use (in German)](https://www.transfermarkt.com/intern/anb)

The basic steps are the same as before but I’ve found that it can be  
quite tricky to find the right nodes on transfermarkt even with the  
CSS Selector Gadget or other methods we described in previous sections.  
After a while you’ll get used to the quirks of how the website is  
structured and know what certain assets (tables, columns, images) are  
called easily. This is a website where the SelectorGadget really comes  
in handy!

This time around I won’t be grabbing an entire table like I did with  
Wikipedia but a number of elements from the webpage. You definitely  
**can** scrape for the table like I showed above with html\_table() but  
in this case I didn’t because the table output was rather messy, gave me  
way more info than I actually needed, and I wasn’t very good at  
regex/stringr to clean the text 2 years ago. Try doing it the way below  
and also by grabbing the entire table for more practice.

The way I did it back then also works out for this blog post because I  
can show you a few other html\_\*() {rvest} functions:

* html\_table(): Get data from a HTML table
* html\_text(): Extract text from HTML
* html\_attr(): Extract attributes from HTML ("src" for image  
  filename, "href" for URL link address)

team\_name <- scrape(session) %>%

html\_nodes("#yw1 > table > tbody > tr > td.zentriert.no-border-rechts > a > img") %>%

html\_attr("alt")

# average age

avg\_age <- scrape(session) %>%

html\_nodes("tbody .hide-for-pad:nth-child(5)") %>%

html\_text()

# average value

avg\_value <- scrape(session) %>%

html\_nodes("tbody .rechts+ .hide-for-pad") %>%

html\_text()

# team image

team\_img <- scrape(session) %>%

html\_nodes("#yw1 > table > tbody > tr > td.zentriert.no-border-rechts > a > img") %>%

html\_attr("src")

With each element collected we can put them into a list and reshape it  
into a nice data frame.

# combine above into one list

resultados <- list(team\_name, avg\_age, avg\_value, team\_img)

# specify column names

col\_name <- c("team", "avg\_age", "avg\_value", "img")

# Combine into one dataframe

j\_league\_age\_value\_raw <- resultados %>%

reduce(cbind) %>%

tibble::as\_tibble() %>%

set\_names(col\_name)

glimpse(j\_league\_age\_value\_raw)

## Rows: 18

## Columns: 4

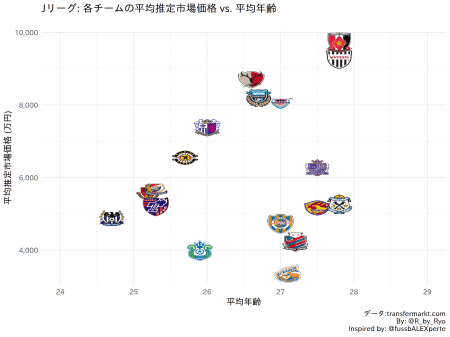
## $ team "Vissel Kobe", "Urawa Red Diamonds", "Kawasaki Frontale",...

## $ avg\_age "25.9", "26.3", "25.5", "24.1", "25.4", "25.0", "25.0", "...

## $ avg\_value "€1.02m", "€698Th.", "€577Th.", "€477Th.", "€524Th.", "€5...

## $ img "<https://tmssl.akamaized.net/images/wappen/tiny/3958.png?.>..

With some more cleaning and {ggplot2} magic you will then get:



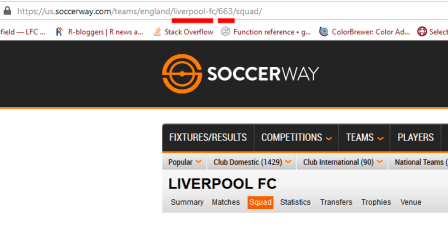
**Multiple Web-pages (Soccerway, Transfermarkt, etc.)**

The previous examples looked at scraping from a single web page but  
usually you want to collect data for each team in a league, each player  
from each team, or each player from each team in every league, etc. This  
is where the added complexity of web-scraping multiple pages comes in.  
The most efficient way is to be able to programatically scrape across  
multiple pages in one go instead of running the same scraping function  
on different teams’/players’ URL link over and over again.

**Thinking About How to Scrape**

* Understand the website structure: How it organizes its pages, check  
  out what the CSS Selector/XPaths are like, etc.
* Get a list of links: Team page links from league page, player page  
  links from team page, etc.
* Create your own R functions: Pinpoint exactly what you want to  
  scrape as well as some cleaning steps post-scraping in one function  
  or multiple functions.
* Start small, then scale up: Test your scraping function on one  
  player/team, then do entire team/league.
* Iterate over a set of URL links: Use {purrr}, for loops,  
  lapply() (whatever your preference).

Look at the URL link for each web page you want to gather. What are the  
similarities? What are the differences? If it’s a proper website than  
the web page for a certain data view for each team should be exactly the  
same, as you’d expect it to contain exactly the same type of info just  
for a different team. For this example each “squad view” page for each  
Premier League team on [soccerway.com](http://soccerway.com) are structured similarly:  
“<https://us.soccerway.com/teams/england/>”  
and then the “team name/”, the “team number/” and finally the name of  
the web page, “squad/”. So what we need to do here is to find out the  
“team name” and “team number” for each of the teams and store them. We  
can then feed each pair of these values in one at a time to scrape the  
information for each team.



url <- "<https://us.soccerway.com/national/england/premier-league/20182019/regular-season/r48730/>"

session <- bow(url)

session

## <https://us.soccerway.com/national/england/premier-league/20182019/regular-season/r48730/>

## User-agent: polite R package -

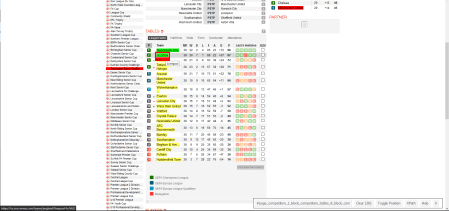
## robots.txt: 4 rules are defined for 3 bots

## Crawl delay: 5 sec

## The path is scrapable for this user-agent

* [Terms of Use](https://us.soccerway.com/terms/)

To find these elements we could just click on the link for each team and  
jot them down … but wait we can just scrape those too! We use the  
html\_attr() function to grab the “href” part of the HTML, which  
contains the hyperlink of that element. The left picture is looking at  
the URL link of one of the buttons to a team’s page via “Inspect”. The  
right picture is selecting every team’s link via the SelectorGadget.



team\_links <- scrape(session) %>%

html\_nodes("#page\_competition\_1\_block\_competition\_tables\_8\_block\_competition\_league\_table\_1\_table .large-link a") %>%

html\_attr("href")

team\_links[[1]]

## [1] "/teams/england/manchester-city-football-club/676/"

The URL given in the href of the HTML for the team buttons  
unfortunately aren’t the **full** URL needed to access these pages. So  
we have to cut out the important bits and re-create them ourselves. We  
can use the {glue} package to combine the “team\_name” and “team\_num”  
for each team in the incomplete URL into a complete URL in a new column  
we’ll call link.

team\_links\_df <- team\_links %>%

tibble::enframe(name = NULL) %>%

## separate out each component of the URL by / and give them a name

tidyr::separate(value, c(NA, NA, NA, "team\_name", "team\_num"), sep = "/") %>%

## glue together the "team\_name" and "team\_num" into a complete URL

mutate(link = glue("[https://us.soccerway.com/teams/england/{team\_name}/{team\_num}/squad/](https://us.soccerway.com/teams/england/%7Bteam_name%7D/%7Bteam_num%7D/squad/)"))

glimpse(team\_links\_df)

## Rows: 20

## Columns: 3

## $ team\_name "manchester-city-football-club", "liverpool-fc", "chelsea...

## $ team\_num "676", "663", "661", "675", "660", "662", "680", "674", "...

## $ link "<https://us.soccerway.com/teams/england/manchester-city-.>..

Fantastic! Now we have the proper URL links for each team. Next we have  
to actually look into one of the web pages itself to figure out what  
exactly we need to scrape from the web page. This assumes that each web  
page and the CSS Selector for the various elements we want to grab are  
the same for every team. As this is for a very simple goal contribution  
plot all we need to gather from each team’s page is the “player name”,  
“number of goals”, and “number of assists”. Use the Inspect element or  
the SelectorGadget tool to grab the HTML code for those stats.

Below, I’ve split each into its own mini-scraper function. When you’re  
working on this part, you should try to use the URL link from one team  
and build your scraper functions from that link (I usually use Liverpool  
as my test example when scraping Premier League teams). Note that all  
three of the mini-functions below could just be chucked into one large  
function but I like keeping things compartmentalized.

player\_name\_info <- function(session) {

player\_name\_info <- scrape(session) %>%

html\_nodes("#page\_team\_1\_block\_team\_squad\_3-table .name.large-link") %>%

html\_text()

}

num\_goals\_info <- function(session) {

num\_goals\_info <- scrape(session) %>%

html\_nodes(".goals") %>%

html\_text()

## first value is blank so remove it

num\_goals\_info\_clean <- num\_goals\_info[-1]

}

num\_assists\_info <- function(session) {

num\_assists\_info <- scrape(session) %>%

html\_nodes(".assists") %>%

html\_text()

## first value is blank so remove it

num\_assists\_info\_clean <- num\_assists\_info[-1]

}

Now that we have scrapers for each stat, we can combine these into a  
larger function that will then gather them all up into a nice data frame  
for each team that we want to scrape. If you input any one of the team  
URLs from team\_links\_df, it will collect the “player name”, “number of  
goals”, and “number of assists” for that team.

premier\_stats\_info <- function(link, team\_name) {

team\_name <- rlang::enquo(team\_name)

## `bow()` for every URL link

session <- bow(link)

## scrape different stats

player\_name <- player\_name\_info(session = session)

num\_goals <- num\_goals\_info(session = session)

num\_assists <- num\_assists\_info(session = session)

## combine stats into a data frame

resultados <- list(player\_name, num\_goals, num\_assists)

col\_names <- c("name", "goals", "assists")

premier\_stats <- resultados %>%

reduce(cbind) %>%

as\_tibble() %>%

set\_names(col\_names) %>%

mutate(team = !!team\_name)

## A little message to keep track of how the function is progressing:

# cat(team\_name, " done!")

return(premier\_stats)

}

**Iteration Over a Set of Links**

OK, so now we have a function that can scrape the data for **ONE** team  
but it would be extremely ponderous to re-run it another NINETEEN times  
for all the other teams… so what can we do? This is where the  
purrr::map() family of functions and iteration comes in! The map()  
family of functions allows you to apply a function (an existing one from  
a package or one that you’ve created yourself) to each element of a list  
or vector that you pass as an argument to the mapping function. For our purposes, this  
means we can use mapping functions to pass along a list of URLs (for  
whatever number of players and/or teams) along with a scraping function  
so that it scrapes it altogether in one go.

In addition, we can use purrr::safely() to wrap any function  
(including custom made ones). This makes these functions return a list  
with the components result and error. This is extremely useful for  
debugging complicated functions as the function won’t just error out and  
give you nothing, but at least the result of the parts of the function  
that worked in result with what didn’t work in error.

So for example, say you are scraping data from the webpage of each team  
in the Premier League (by iterating a single scraping function over each  
teams’ web page) and by some weird quirk in the HTML of the web page or  
in your code, the data from one team errors out (while the other 19  
teams’ data are gathered without problems). Normally, this will mean the  
data you gathered from **all** other web pages that **did** work  
**won’t** be returned, which can be extremely frustrating. With a  
safely() wrapped function, the data from the 19 teams’ data that the  
function was able to scrape is returned in result component of the  
list object while the one errored team and error message is returned in  
the error component. This makes it very easy to debug when you know  
exactly which iteration of the function failed.

safe\_premier\_stats\_info <- safely(premier\_stats\_info)

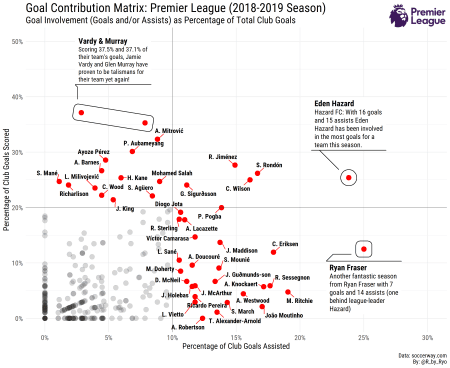
We already have a nice list of team URL links in the data frame  
team\_links\_df, specifically in the “link” column  
(team\_links\_df$link). So we pass that along as an argument to map2()  
(which is just a version of map() but for two argument inputs) and our  
premier\_stats\_info() function so that the function will be applied to  
**each** team’s URL link. This part may take a while depending on your  
internet connection and/or if you put a large value for the crawl delay.

goal\_contribution\_df\_ALL <- map2(.x = team\_links\_df$link, .y = team\_links\_df$team\_name,

~ safe\_premier\_stats\_info(link = .x, team\_name = .y))

## check out the first 4 results:

glimpse(head(goal\_contribution\_df\_ALL, 4))



As you can see (the results/errors for the first four teams scraped),  
for each team there is a list holding a “result” and “error” element.  
For the first four, at least, it looks like everything was scraped  
properly into a nice data.frame. We can check if any of the twenty teams  
had an error by purrr::discard()-ing any elements of the list that  
come out as NULL and seeing if there’s anything left.

## check to see if any failed:

goal\_contribution\_df\_ALL %>%

map("error") %>%

purrr::discard(~is.null(.))

## list()

It comes out as a empty list which means were no errors in the “error”  
elements. Now we can squish and combine individual team data.frames into  
one data.frame using dplyr::bind\_rows().

goal\_contribution\_df <- goal\_contribution\_df\_ALL %>%

map("result") %>%

bind\_rows()

glimpse(goal\_contribution\_df)

## Rows: 622

## Columns: 4

## $ name "C. Bravo", "Ederson Moraes", "S. Carson", "K. Walker", "J....

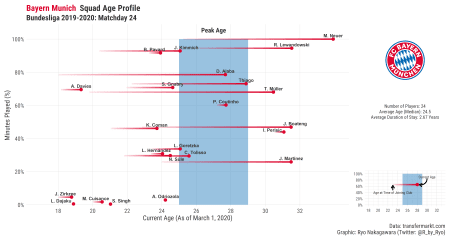
## $ goals "0", "0", "0", "1", "0", "0", "0", "0", "0", "2", "0", "0",...

## $ assists "0", "0", "0", "2", "0", "0", "0", "2", "0", "0", "0", "0",...

## $ team "manchester-city-football-club", "manchester-city-football-...

With that we can clean the data a bit and finally get on to the plotting!

For practice, try doing it for a different season or for a different  
league altogether!



**Conclusion**

This blog post went over web-scraping, focusing on getting soccer data  
from soccer websites in a responsibly fashion. After a brief overview of  
responsible scraping practices with R I went over several examples of  
getting soccer data from various websites. I make no claims that its the  
most efficient way, but importantly, it gets the job done and in a  
polite way. More industrial-scale scraping over hundreds and thousands  
of web pages is a bit out of scope for an introductory blog post and  
it’s not something I’ve really done either, so I will pass along the  
torch to someone else who wants to write about that. There are other  
ways to scrape websites using R, especially websites that have dynamic  
web pages

**Happy (responsible) Web-scraping!**