An Introduction to Web Scraping in R

Last Updated: 2021-08-15

Note: This document is meant for internal uses within UCSB. Proper citations have not been included in this draft. Please use this document as an individual learning tool, not for official citation or distribution.

```
# Packages for lesson:
library(pacman)
pacman::p_load(
  "jsonlite",# read in json formats
  "formatR", # makes code wrap in an Rmarkdown file
  "glue", # tidyverse concatenate text
  "magrittr",# fancy pipe
  "polite",# make sure website is okay to scrape
  "rvest",# Grabbing, formatting, and cleaning html code
  "tidyverse"# Thanks Hadley Wickham!
)
```

In addition, this lesson will require using Google Chrome and the selector gadget extension. This link is to Google Chrome selector gadget extension. Please add the selector gadget to Google Chrome now.

1 What is Web Scraping?

Web scraping is a general term that tends to relate to pulling data from some web pages and converting it into a usable, or tidy, form. Colloquially, we tend to refer to two different methods when we say "web scraping".

The first is actually **API pulling**. This is accessing a website's underlying data organization and directly using that. In this case, the data is in a clean and usable form - we just have to pull it in. The second is actual **web scraping**. This process involves pulling html code from a website, cleaning and ultimately creating a tidy dataset. If given the choice, always API pull. API pulling is easier, there is less chance for user error, and ultimately faster.

In this lesson, we'll go through an example of pulling from an API and web scraping. We'll be pulling from the Hechinger Report of UC - Santa Barbara as an example of an API pull and scraping the UCSB Economics directory. For the purposes of this lesson, we'll use the term *scrape* to describe both pulling and scraping from a website (as is colloquial).

Copyright UCSB 2021

2 Can I Scrape From This Site?

Before scraping a website, we must ensure the site is scrape-able. This means that the owners of the site are okay with individuals taking information from their site. Reasons they may not be okay with this is proprietary information or crashing the site. Constantly accessing sites with a scraper, or code meant to extract data from a website, can lead to performance issues and ultimately crashing a site. To see if we're allowed to scrape from a site, we'll use the package `polite`. There is a useful repository here.

`polite` 's main function is `bow`.[1] This command is politely asking if we have permission to access a website. The general framework of the command is as follows:

```
bow(
  url = "https://tuitiontracker.org/fitness/data/school-data/",

  user_agent = "Danny Klinenberg <dklinenberg@ucsb.edu>"
)
```

The command `url` is our target site we wish to scrape. `user_agent` is our name and email. While the `user_agent` is not necessary, it's extra polite to the hosts of the website.

If we're allowed to scrape from a website, the result will look like this:

```
bow(
  url = "https://tuitiontracker.org/fitness/data/school-data/",

  user_agent = "Danny Klinenberg <dklinenberg@ucsb.edu>"
)
```

```
## <polite session> https://tuitiontracker.org/fitness/data/school-data/
##     User-agent: Danny Klinenberg <dklinenberg@ucsb.edu>
##     robots.txt: 1 rules are defined for 1 bots
##    Crawl delay: 5 sec
##   The path is scrapable for this user-agent
```

Let's walk through the lines of output. Line one is the website we're asking to pull from. Line 2 is us identifying ourselves. Line 3 are the rules of pulling from their website.[2] Line 4 is the crawl delay. This is how much time the hosts requests between pulling from the website. Hosts asks for crawl delays to avoid overloading the site leading to a crash. If we are to pull alot of time from a website, we may get blocked for not following this rule. We'll talk about how to add time between pulls in the UCSB Economics example.[3]

If we are not allowed to scrape, the results will look like this:

[1] The whole package is inspired by British etiquette at a tea party. They really get into it on their blogpost.

[2] I'm not sure how to access the rules and it hasn't come up as an issue yet so hopefully it's fine?

[3] For those extra excited, we'll be using `Sys.Sleep` at the beginning of each loop.

Copyright UCSB 2021

```
bow(
  url = "https://collegecrisis.shinyapps.io/dashboard/",

  user_agent = "Danny Klinenberg <dklinenberg@ucsb.edu>"
)
```

```
## <polite session> https://collegecrisis.shinyapps.io/dashboard/
##     User-agent: Danny Klinenberg <dklinenberg@ucsb.edu>
##     robots.txt: 1 rules are defined for 1 bots
##    Crawl delay: 5 sec
##   The path is not scrapable for this user-agent
```

The legal, ethical, and moral issues of scraping web pages are a popular topic right now. For example, are we really not allowed to scrape a site that asks us not to scrape? Can the authors really do anything to us? If we find ourselves in need of information on a site we're not given permission to scrape, the best next step is to contact the authors of the site. It's safest and you don't want to find yourself in a legal battle over some data. It's (probably) not worth it?

3 Pulling from an API: Hechinger's Report

The Hechinger's Report created a college financial fitness report for most universities based off of the book *College Stress Test*. The report shows the financial health of a school along four metrics: enrollment, retention, average tuition, and appropriations. Let's take a look at how UCSB did. The url for UCSB's financial health report is https://tuitiontracker.org/fitness/school.html?unitid=110705. The page looks like this (I added the green circle):

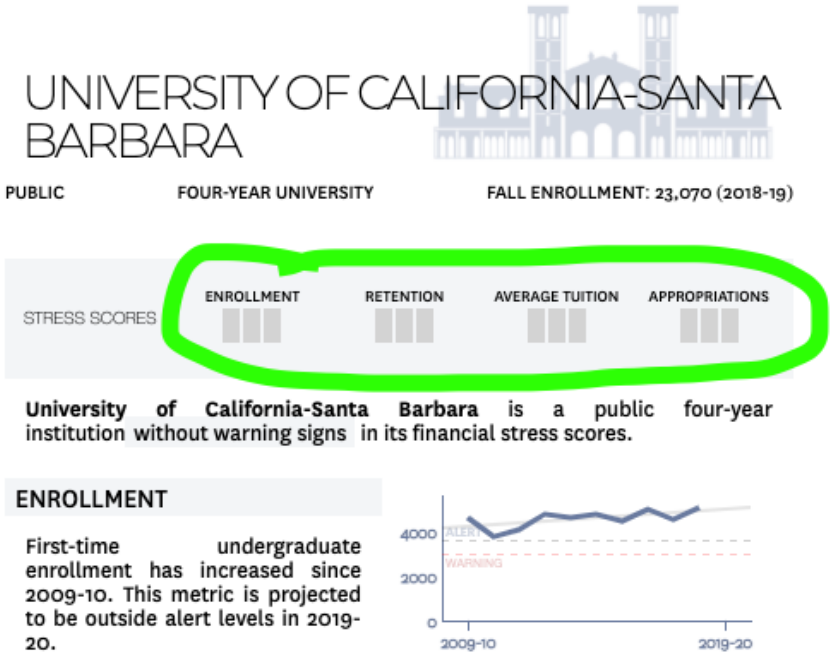

Figure 1: UCSB Report

We're interested in getting the scores for the four stress variables. Bars filled in with red signify levels of stress. More red bars means more stress. In UCSB's case, they have no stress (yay!). However, this isn't in a format to pull directly into a tibble. Analysis is not possible. . . yet.

If we're using Google chrome, we can dive into the website source code by right clicking on the page and choosing `inspect`:

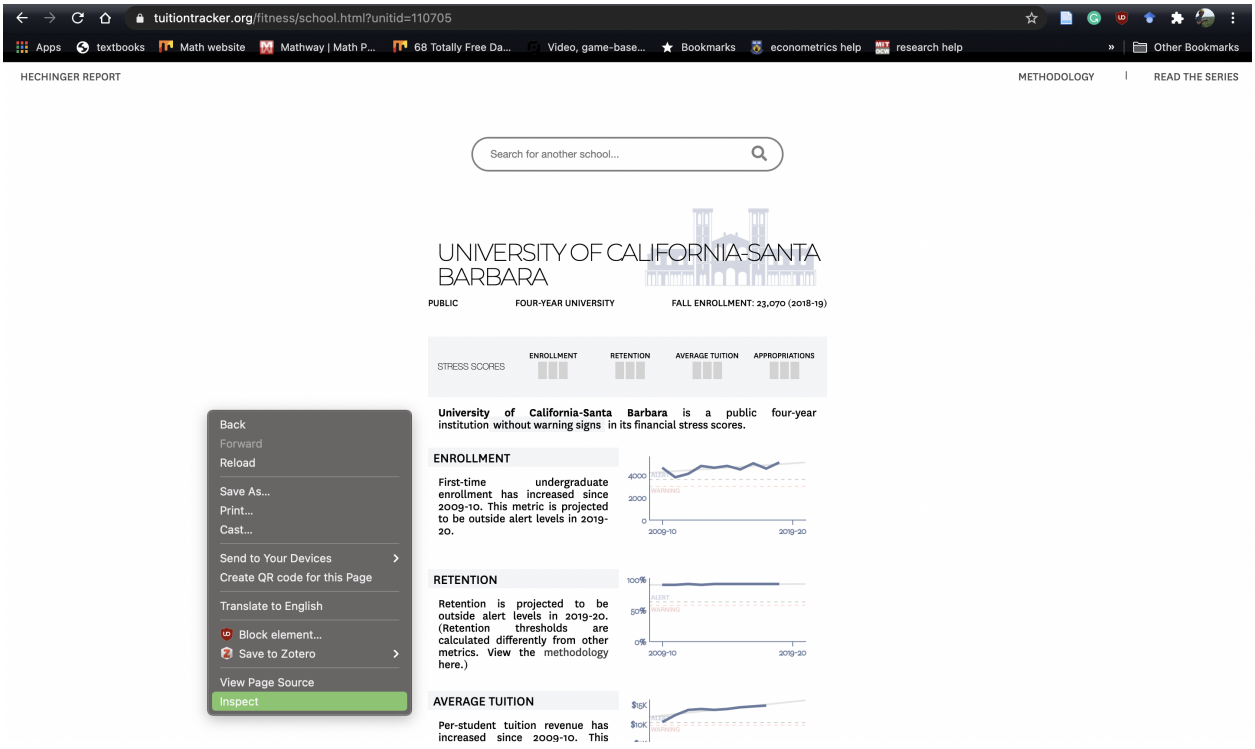

Figure 2: Opening Inspector

After opening inspector, we immediately see half our screen is engulfed with funky code. This is html code, the bones of the website:

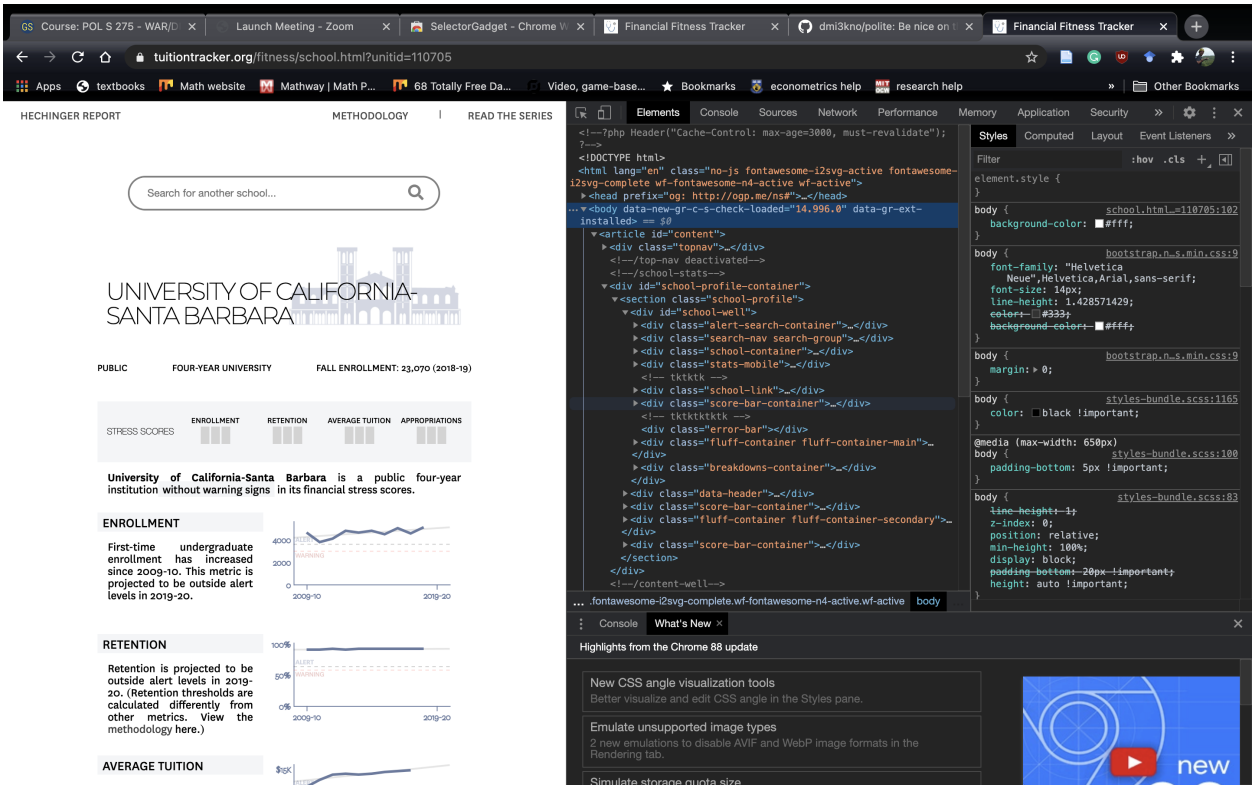

Figure 3: HTML Code

As a fun tangent, to see which part of the webpage corresponds to which part of the html code we can press `cmnd+shft+c` on a Mac. To exit out of this mode, we can press `esc`.

Most of the time, websites pull data from their own API. To see if this is the case here, we can go to the *network* tab:

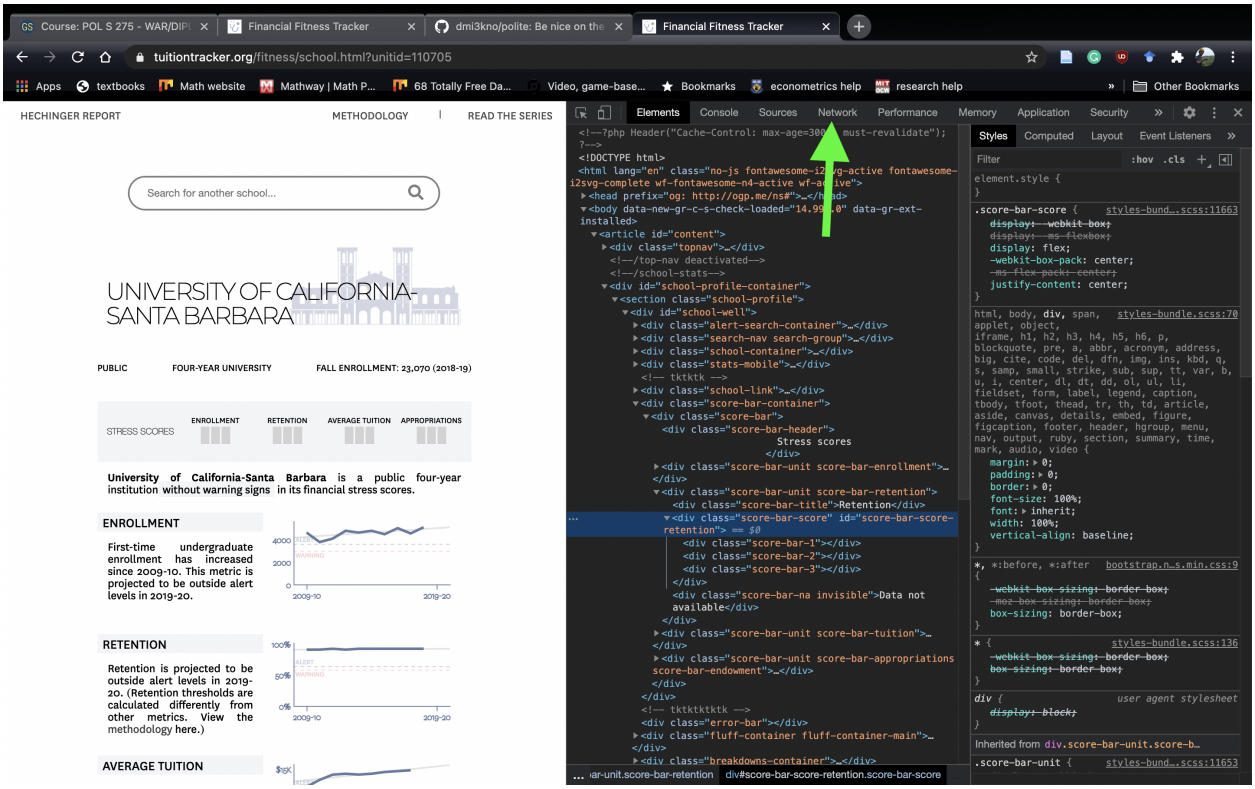

Figure 4: Press the Network tab: Green Arrow

Now that we have the network tab open, the key is to refresh the page. That way, we'll be able to see all the files this web page pulls in. And with a bit of luck, we'll find the file that houses all the data we need.

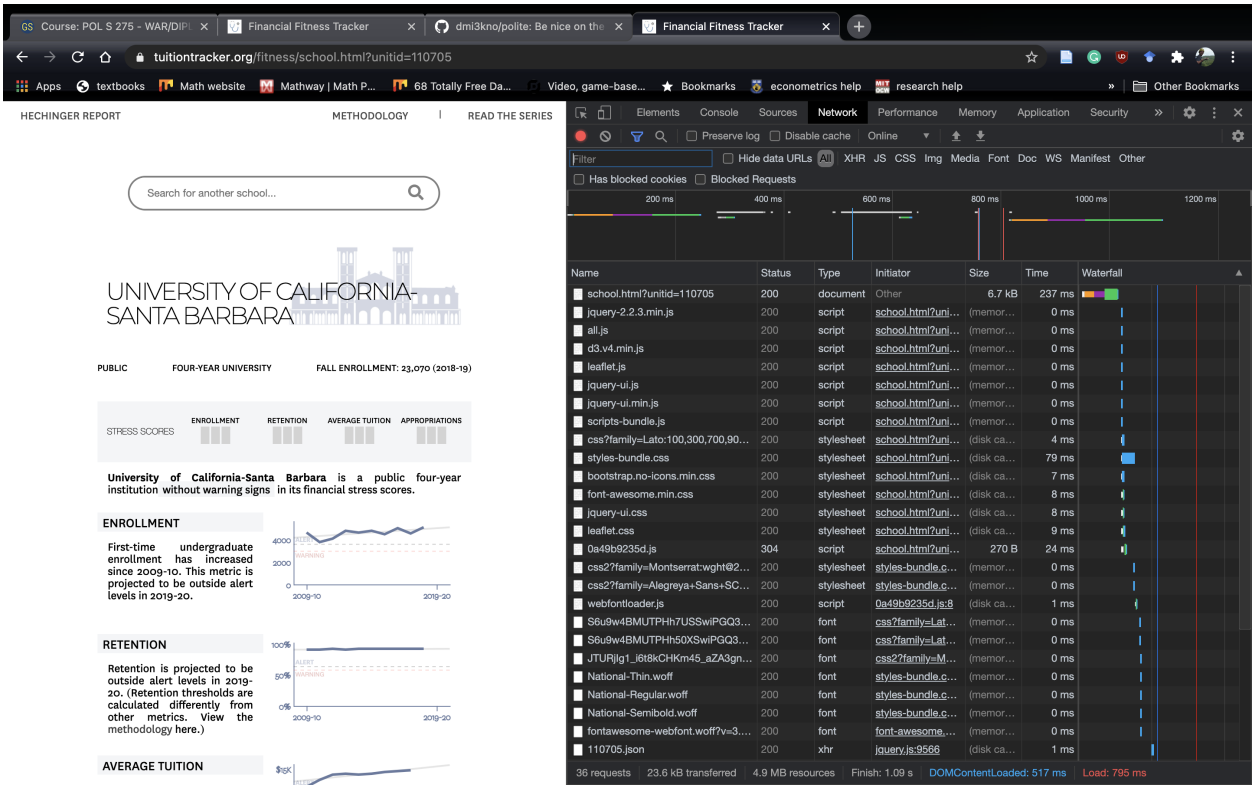

Figure 5: Network Tab After Refreshing the Page

Most of these files are well beyond my understanding of website infrastructure. However, I have figured out that sites tend to store their underlying data as a `.csv` , `.json` or `.ndjson` . A `.json` is a more flexible `.csv`. Think of a `.json` as a `.csv` only instead of each cell housing a single data, it can house another `.csv`. In `R` terms, a `.json` is like a list. A `.ndjson` is like a `.json` only each row is it's very own `.json`.[4] Once we find a .`json`, we can right click it and copy the file path. Here, I'm opening **110705.json**:

[4]You'll have to work with `.ndjson` files if you want to use the Parler data dump.

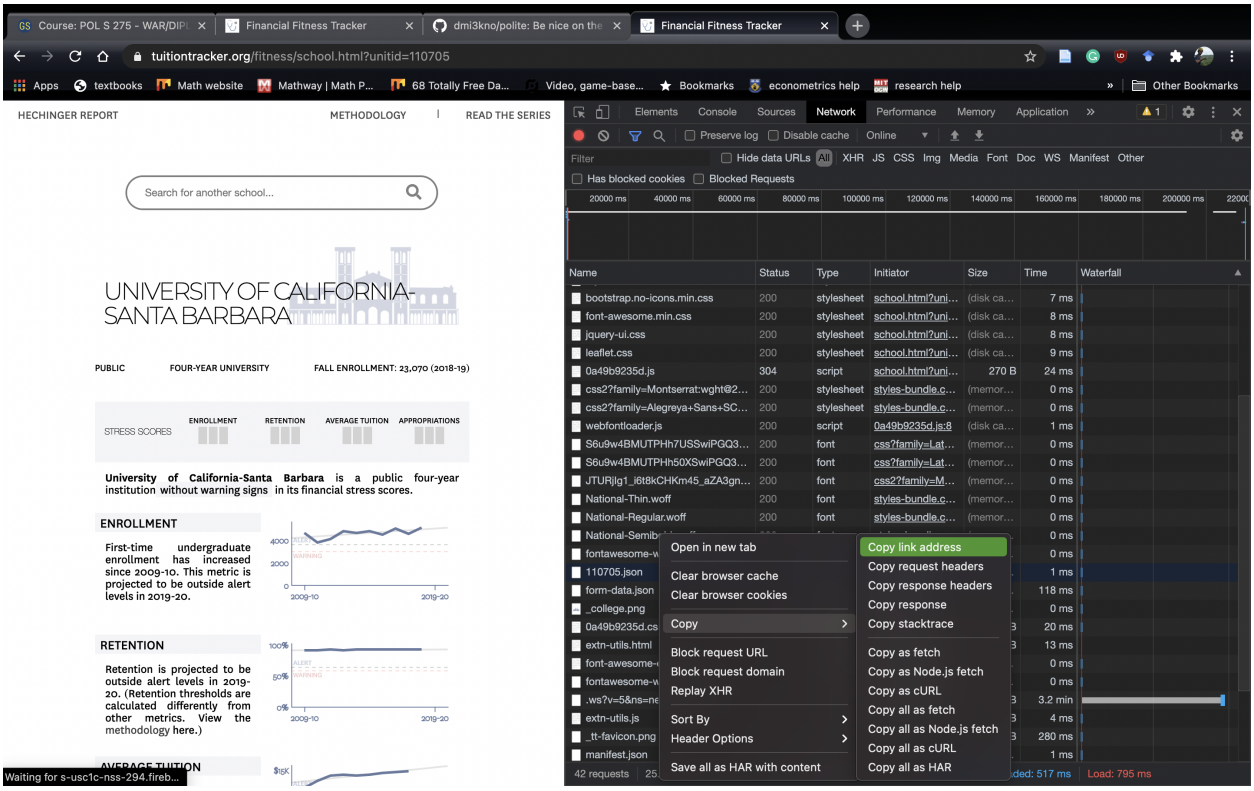

Figure 6: Opening .json Attempt 1

Once we copy the link address, we can paste it in a new tab in Chrome and see where it takes us:

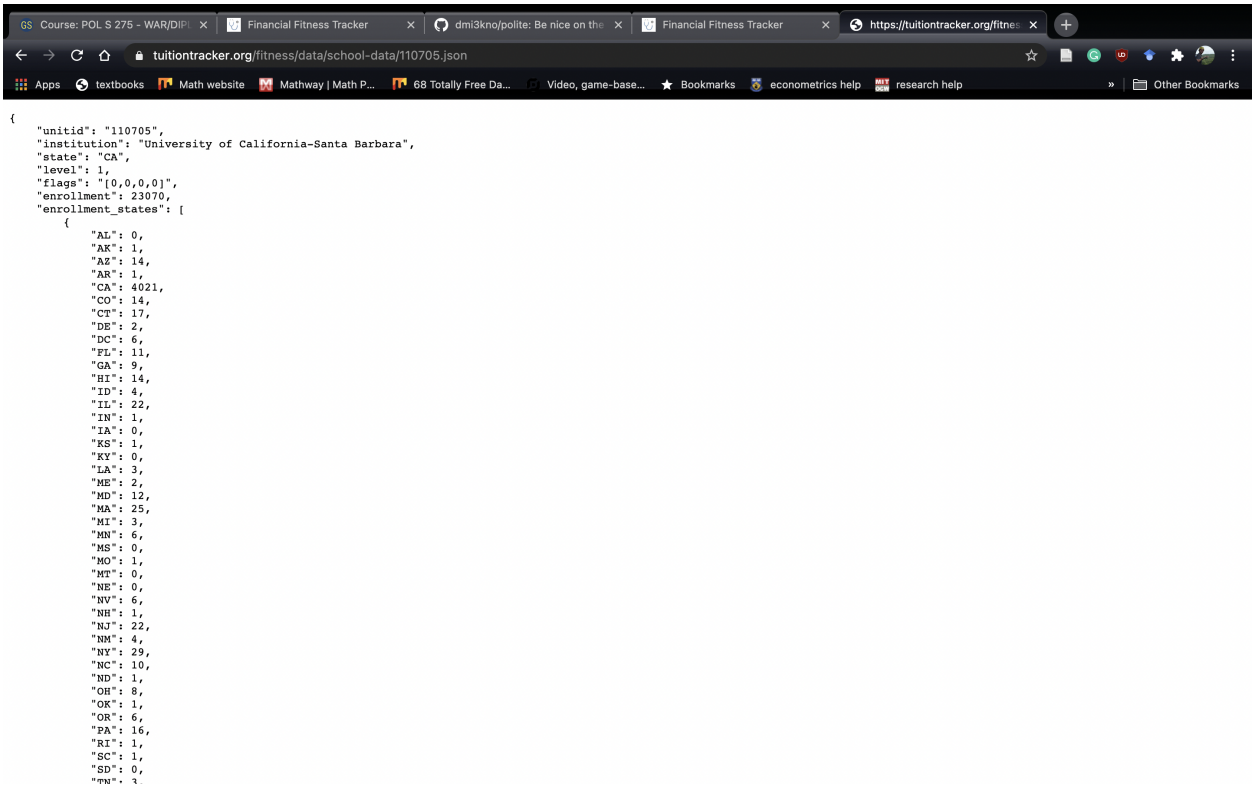

Figure 7: Opening the .json Link

We grabbed the right one. Nice! We can then import the file directly into R:

```
ucsb <- jsonlite::read_json("https://tuitiontracker.org/fitness/data/school-data/110705.
```

We can then grab things like the name, IPEDS UNITID, and the stress scores (stored in flags):

```
ucsb$institution
```

```
## [1] "University of California-Santa Barbara"
```

```
ucsb$unitid
```

```
## [1] "110705"
```

```
ucsb$flags
```

```
## [1] "[0,0,0,0]"
```

Now, having data on one school is cool, but what if we had data on all the schools. For example, suppose we were interested in finding the stress scores for the following schools:

```
schools <- tibble(schools = c("UCSB", "Cal Poly SLO", "Sac State"),
    unitid = c(110705, 110422, 110617))
print(schools)
```

```
## # A tibble: 3 x 2
##   schools      unitid
##   <chr>         <dbl>
## 1 UCSB         110705
## 2 Cal Poly SLO 110422
## 3 Sac State    110617
```

Let's look at the url for UCSB: "https://tuitiontracker.org/fitness/data/school-data/110705.json%22.

Now, let's look at the url for Sac State: "https://tuitiontracker.org/fitness/data/school-data/110617.json"

Notice that the urls are identical except of the numbers before .json. Usually, websites have some sort of method for their url names. In this case, the data url is "https://tuitiontracker.org/fitness/data/school-data/**UNITID**.json", where UNITID is the IPEDS UNITID. We can pull those off of IPEDS easy, so all we'd need to do to get the stress scores is loop through the UNITIDS! Let's do that here:

```
num_schools <- 3  # interested in 3 schools
dat <- tibble(school = rep(NA, num_schools), unitid = rep(NA,
    num_schools), enrollment_stress = rep(NA, num_schools),
    retention_stress = rep(NA, num_schools), avg_tui_stress = rep(NA,
        num_schools), approp_stress = rep(NA, num_schools))

for (i in 1:num_schools) {
    # remember, we're polite, so we honor the 5 second
    # rule
    Sys.sleep(5)
    school <- jsonlite::read_json(paste0("https://tuitiontracker.org/fitness/data/school
        schools$unitid[i], ".json"))
    # clean the flag scores and unlist
    scores <- school$flags %>%
        str_replace_all("\\[|\\]", "") %>%
        str_split(",") %>%  str_split(",") %>%
    unlist()
    # save the stuff
    dat$school[i] <- school$institution
```

```
    dat$unitid[i] <- school$unitid
    dat$enrollment_stress[i] <- scores[1]
    dat$retention_stress[i] <- scores[2]
    dat$avg_tui_stress[i] <- scores[3]
    dat$approp_stress[i] <- scores[4]
}
```

```
## Warning in stri_split_regex(string, pattern, n = n, simplify = simplify, :
## argument is not an atomic vector; coercing

## Warning in stri_split_regex(string, pattern, n = n, simplify = simplify, :
## argument is not an atomic vector; coercing

## Warning in stri_split_regex(string, pattern, n = n, simplify = simplify, :
## argument is not an atomic vector; coercing
```

```
dat
```

```
## # A tibble: 3 x 6
##   school   unitid enrollment_stre~ retention_stress avg_tui_stress approp_stress
##   <chr>    <chr>  <chr>            <chr>            <chr>          <chr>
## 1 Univers~ 110705 "c(\"0\""        " \"0\""         " \"0\""       " \"0\")"
## 2 Califor~ 110422 "c(\"0\""        " \"0\""         " \"0\""       " \"0\")"
## 3 Califor~ 110617 "c(\"0\""        " \"0\""         " \"3\""       " \"0\")"
```

To recap, we performed the following steps:

1. Identified a website we wanted to scrape
2. Used the `polite` package to see if we were allowed to scrape the site
3. We used Google Chrome's inspector function to look at the source code
4. We went to network and refreshed the page to try and find the source of the data
5. We guessed which file had the data. When we found it, we imported the data into `R` using the url.

4 Web Scraping: UCSB Economics Contact Information

Pulling from an API is always the preferable approach. However, sometimes we can't pull from an API because it isn't publicly available. In that case, we have to scrape the html

directly from the website. Hadley Wickham, tidyverse creator, developed the `rvest` to assist in web scraping in R. There are many blog posts describing how to use `rvest`, but I'd recommend starting with the official github page. This example will require using `SelectorGadget`.

In my experience, scraping a web page has a very similar code structure:

```
read_html("website_name") %>% # read in html
  html_nodes("found with selector gadget") %>% #find right node
  html_text() %>% # convert to text
  as_tibble() # tidy it up
```

The first line reads in the website, the second line tells R which part of the html code to find, the third line converts it into readable text, and the fourth line puts it in a tibble. Each line will be discussed in detail through the example.

We're going to pull data from the UCSB Economics Directory.[5] Our goal is to create a tibble with an individual's name.

To start, let's take a look at the website:

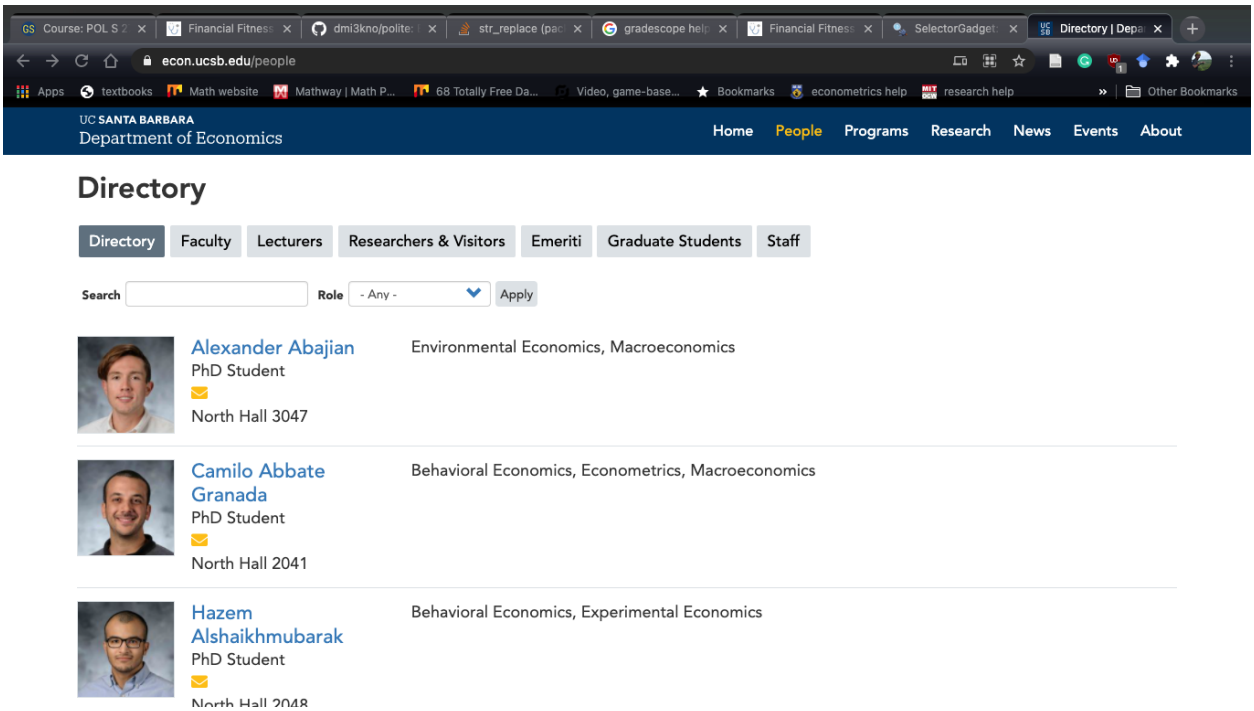

Figure 8: UCSB Economics Directory Screenshot

First, let's make sure we have permission to scrape this site:

[5] If the UCSB page has undergone major revisions, you can replace the url with an archived verison of the page: https://web.archive.org/web/20210722193128/https://econ.ucsb.edu/people

```
bow(url="https://econ.ucsb.edu/people",

    user_agent = "Danny Klinenberg <dklinenberg@ucsb.edu>")
```

```
## <polite session> https://econ.ucsb.edu/people
##     User-agent: Danny Klinenberg <dklinenberg@ucsb.edu>
##     robots.txt: 42 rules are defined for 1 bots
##    Crawl delay: 5 sec
##   The path is scrapable for this user-agent
```

Good news, we have permission! Now, we're going to read in the raw html code into R:

```
website <- read_html("https://econ.ucsb.edu/people")
```

Take a moment to open the website object in R. You should notice that it's incomprehensible. It's some list titled `<html>` then it has something called `<body>` and some other stuff. Thanks to Hadley, we don't need to parse through the raw html code to try and extract our desired information. Instead, we can use the `rvest` package. Through rvest, we can tell `R` which specific node[6] of the html code we want and it'll grab the information for that node. To know which node we want, we use the selector gadget in chrome. To open selector gadget, click the puzzle piece in the top right hand corner, then the magnifying glass:

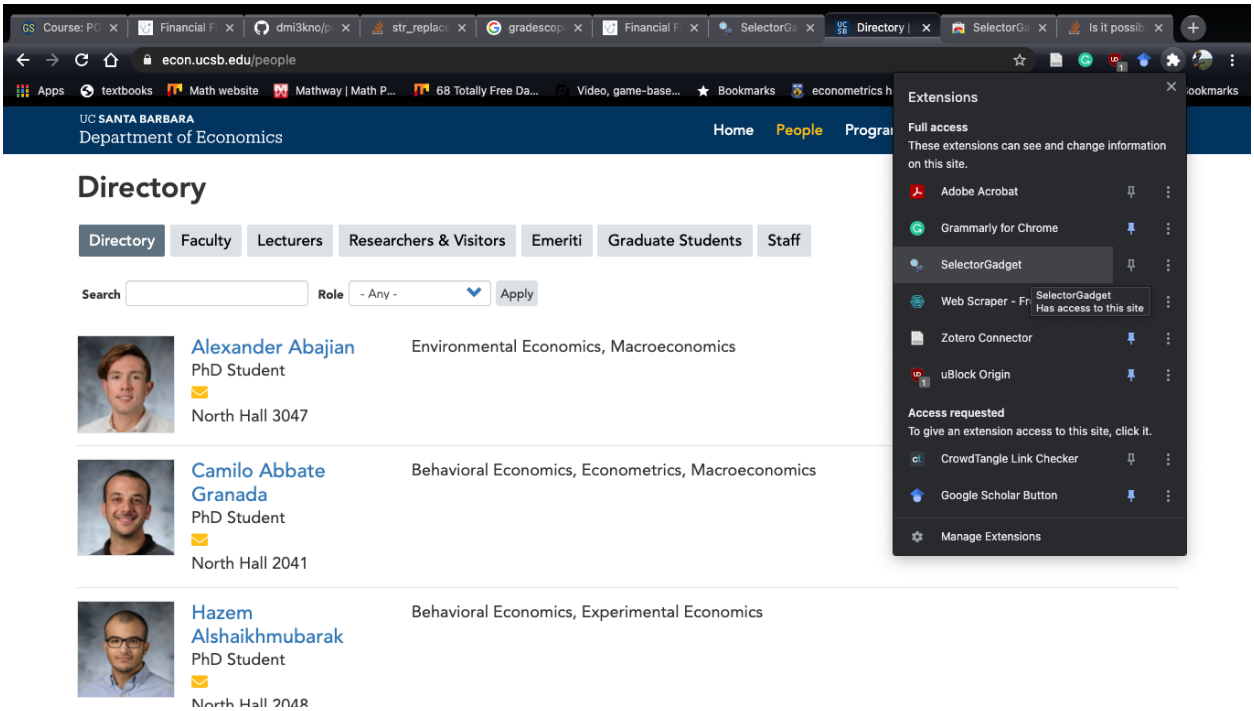

Figure 9: Opening Selector Gadget

[6] html lingo for element.

Once open, you'll notice a bunch of random orange squares appearing when you move your mouse. If you click on/next to a specific element, the square will turn green. At the bottom of the screen, you'll notice a string in a box:

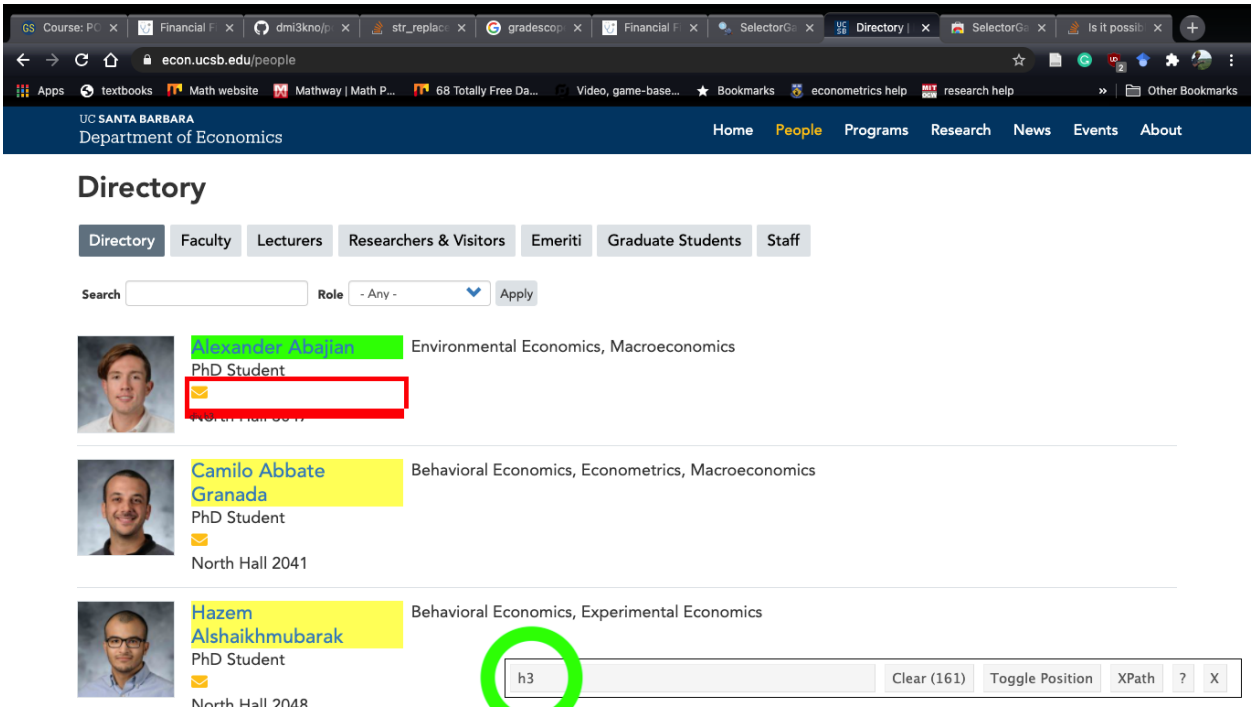

Figure 10: Selector Gadget after clicking next to Alexander Abajian

The green is the html node we asked for while the yellow boxes are the "guesses" selector gadget is making on information you want. If you click a yellow box (e.g. next to Camilo Abbate Granada's name), the box will turn red and the string at the bottom (green circle) will change. If you click Camilo's name again, it will turn yellow again. Once we have the information we want highlighted, we copy the string in the green circle and paste it into `html_nodes`:

```
website %>%
    html_nodes("h3") %>%
    .[1:5]  # only displaying first 5
```

```
## {xml_nodeset (5)}
## [1] <h3><a href="/people/students/alexander-abajian" hreflang="en">Alexander  ...
## [2] <h3><a href="/people/students/camilo-abbate-granada" hreflang="en">Camilo ...
## [3] <h3><a href="/people/students/hazem-alshaikhmubarak" hreflang="en">Hazem  ...
## [4] <h3><a href="/people/students/roberto-amaral-de-castro-prado-santos" href ...
## [5] <h3><a href="/people/lecturers/bob-anderson" hreflang="en">Bob Anderson</ ...
```

We could parse the information we want ourselves, or we could use the `html_text` command:

```
website %>%
  html_nodes("h3") %>%
  html_text(trim=T) %>% # removes hanging white space at the beginning and end
  as_tibble() %>%
  slice(1:5) # only displaying first 5
```

```
## # A tibble: 5 x 1
##   value
##   <chr>
## 1 Alexander Abajian
## 2 Camilo Abbate Granada
## 3 Hazem Alshaikhmubarak
## 4 Roberto Amaral de Castro Prado Santos
## 5 Bob Anderson
```

To extend this, you can add html endpoints to `html_nodes` using inspector gadget. Then it's a matter of parsing the data using our data wrangling tools!

Copyright UCSB 2021