Big Data in Economics

Lecture 8: Webscraping: (2) Client-side and APIs

Grant R. McDermott

University of Oregon | EC 510

Contents

Sign-up and software requirements

Sign-up

We're going to be downloading economic data from the FRED API. This will require that you first create a user account and then register a personal API key.

External software

Today I'll be using JSONView, a browser extension that renders JSON output nicely in Chrome and Firefox. (Not required, but recommended.)

R packages

- New: **jsonlite**, **httr**, **listviewer**, **usethis**, **fredr**
- Already used: **tidyverse**, **lubridate**, **hrbrthemes**, **janitor**

Here's a convenient way to install (if necessary) and load all of the above packages.

```
## Load and install the packages that we'll be using today
if (!require("pacman")) install.packages("pacman")
pacman::p_load(tidyverse, httr, lubridate, hrbrthemes, janitor, jsonlite, listviewer, usethis)
pacman::p_install_gh("sboysel/fredr") ## https://github.com/sboysel/fredr/issues/75
## My preferred ggplot2 plotting theme (optional)
theme_set(hrbrthemes::theme_ipsum())
```

Recap from last time

During the last lecture, we saw that websites and web applications fall into two categories: 1) Server-side and 2) Client-side. We then practised scraping data that falls into the first category — i.e. rendered server-side — using the **rvest** package. This technique focuses on CSS selectors (with help from SelectorGadget) and HTML tags. We also saw that webscraping

often involves as much art as science. The plethora of CSS options and the flexibility of HTML itself means that steps which work perfectly well on one website can easily fail on another website.

Today we focus on the second category: Scraping web data that is rendered **client-side**. The good news is that, when available, this approach typically makes it much easier to scrape data from the web. The downside is that, again, it can involve as much art as it does science. Moreover, as I emphasised last time, just because because we *can* scrape data, doesn't mean that we *should* (i.e. ethical and other considerations). These admonishments aside, let's proceed…

Client-side, APIs, and API endpoints

Recall that websites or applications that are built using a **client-side** framework typically involve something like the following steps:

- You visit a URL that contains a template of static content (HTML tables, CSS, etc.). This template itself doesn't contain any data.
- However, in the process of opening the URL, your browser sends a *request* to the host server.
- If your request if valid, then the server issues a *response* that fetches the necessary data for you and renders the page dynamically in your browser.
- The page that you actually see in your browser is thus a mix of static content and dynamic information that is rendered by your browser (i.e. the "client").

All of this requesting, responding and rendering takes places through the host application's **API** (or **A**pplication **P**rogram **I**nterface).

APIs

If you're completely new to APIs, then I recommend this excellent resource from Zapier: An Introduction to APIs. It's fairly in-depth, but you don't need to work through the whole thing to get the gist. The summary version is that an API is really just a collection of rules and methods that allow different software applications to interact and share information. This includes not only web servers and browsers, but also software packages like the R libraries we've been using.[1] Key concepts include:

- **Server:** A powerful computer that runs an API.
- **Client:** A program that exchanges data with a server through an API.
- **Protocol:** The "etiquette" underlying how computers talk to each other (e.g. HTTP).
- **Methods:** The "verbs" that clients use to talk with a server. The main one that we'll be using is `GET` (i.e. ask a server to retrieve information), but other common methods are `POST`, `PUT` and `DELETE`.
- **Requests:** What the client asks of the server (see Methods above).
- **Response:** The server's response. This includes a *Status Code* (e.g. "404" if not found, or "200" if successful), a *Header* (i.e. meta-information about the reponse), and a *Body* (i.e the actual content that we're interested in).
- Etc.

A bit more about API endpoints

A key point in all of this is that, in the case of web APIs, we can access information *directly* from the API database if we can specify the correct URL(s). These URLs are known as an **API endpoints**.

API endpoints are in many ways similar to the normal website URLs that we're all used to visiting. For starters, you can navigate to them in your web browser. However, whereas normal websites display information in rich HTML content — pictures, cat videos, nice formatting, etc. — an API endpoint is much less visually appealing. Navigate your browser to an API endpoint and you'll just see a load of seemingly unformatted text. In truth, what you're really seeing is (probably) either JSON (**J**ava**S**cript **O**bject **No**tation) or XML (E**x**tensible **M**arkup **L**anguage).

You don't need to worry too much about the syntax of JSON and XML. The important thing is that the object in your browser — that load of seemingly unformatted text — is actually very precisely structured and formatted. Moreover, it

[1] Fun fact: A number of R packages that we'll be using later in this course (e.g. **leaflet**, **plotly**, etc.) are really just a set of wrapper functions that interact with the underlying APIs and convert your R code into some other language (e.g. JavaScript).

contains valuable information that we can easily read into R (or Python, Julia, etc.) We just need to know the right API endpoint for the data that we want.

Let's practice doing this through a few example applications. I'll start with the simplest case (no API key required, explicit API endpoint) and then work through some more complicated examples.

Application 1: Trees of New York City

NYC Open Data is a pretty amazing initiative. Its mission is to "make the wealth of public data generated by various New York City agencies and other City organizations available for public use". You can get data on everything from arrest data, to the location of wi-fi hotspots, to city job postings, to homeless population counts, to dog licenses, to a directory of toilets in public parks… The list goes on. I highly encourage you to explore in your own time, but we're going to do something "earthy" for this first application: Download a sample of tree data from the **2015 NYC Street Tree Census**.

I wanted to begin with an example from NYC Open Data, because you don't need to set up an API key in advance.[2] All you need to do is complete the following steps:

- Open the web page in your browser (if you haven't already done so).
- You should immediately see the **API** tab. Click on it.
- Copy the API endpoint that appears in the popup box.
- *Optional:* Paste that endpoint into a new tab in your browser. You'll see a bunch of JSON text, which you can render nicely using the JSONView browser extension that we installed earlier.

Here's a GIF of me completing these steps:

```
## Sorry, this GIF only available in the the HTML version of the notes.
```

Now that we've located the API endpoint, let's read the data into R. We'll do so using the `fromJSON()` function from the excellent **jsonlite** package (link). This will automatically coerce the JSON array into a regular R data frame. However, I'll go that little bit further and convert it into a tibble, since the output is nicer to work with.

```
# library(jsonlite) ## Already loaded

nyc_trees <-
  fromJSON("https://data.cityofnewyork.us/resource/nwxe-4ae8.json") %>%
  as_tibble()
nyc_trees
```

```
## # A tibble: 1,000 x 45
##    tree_id block_id created_at tree_dbh stump_diam curb_loc status health
##    <chr>   <chr>    <chr>      <chr>    <chr>      <chr>    <chr>  <chr>
##  1 180683  348711   2015-08-2~ 3        0          OnCurb   Alive  Fair
##  2 200540  315986   2015-09-0~ 21       0          OnCurb   Alive  Fair
##  3 204026  218365   2015-09-0~ 3        0          OnCurb   Alive  Good
##  4 204337  217969   2015-09-0~ 10       0          OnCurb   Alive  Good
##  5 189565  223043   2015-08-3~ 21       0          OnCurb   Alive  Good
##  6 190422  106099   2015-08-3~ 11       0          OnCurb   Alive  Good
##  7 190426  106099   2015-08-3~ 11       0          OnCurb   Alive  Good
##  8 208649  103940   2015-09-0~ 9        0          OnCurb   Alive  Good
##  9 209610  407443   2015-09-0~ 6        0          OnCurb   Alive  Good
## 10 192755  207508   2015-08-3~ 21       0          OffsetF~ Alive  Fair
## # … with 990 more rows, and 37 more variables: spc_latin <chr>,
## #   spc_common <chr>, steward <chr>, guards <chr>, sidewalk <chr>,
## #   user_type <chr>, problems <chr>, root_stone <chr>, root_grate <chr>,
## #   root_other <chr>, trunk_wire <chr>, trnk_light <chr>, trnk_other <chr>,
## #   brch_light <chr>, brch_shoe <chr>, brch_other <chr>, address <chr>,
```

[2] Truth be told: To avoid rate limits — i.e. throttling the number of requests that you can make per hour — it's best to sign up for an NYC Open Data app token. We're only going to make one or two requests here, though so we should be fine.

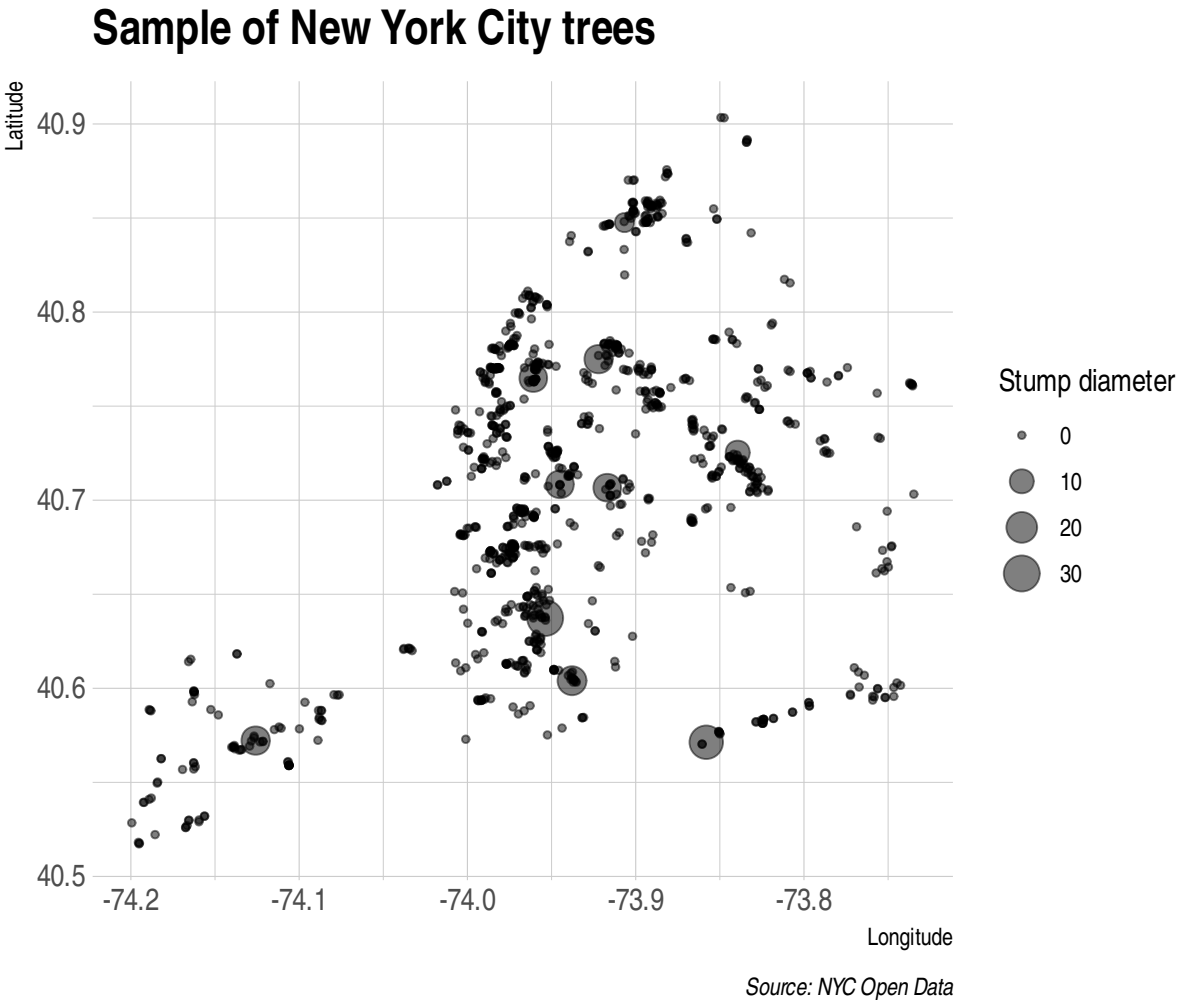
```
## #   zipcode <chr>, zip_city <chr>, cb_num <chr>, borocode <chr>,
## #   boroname <chr>, cncldist <chr>, st_assem <chr>, st_senate <chr>, nta <chr>,
## #   nta_name <chr>, boro_ct <chr>, state <chr>, latitude <chr>,
## #   longitude <chr>, x_sp <chr>, y_sp <chr>, council_district <chr>,
## #   census_tract <chr>, bin <chr>, bbl <chr>
```

Aside on limits: Note that the full census dataset contains nearly 700,000 individual trees. However, we only downloaded a tiny sample of that, since the API defaults to a limit of 1,000 rows. I don't care to access the full dataset here, since I just want to illustrate some basic concepts. Nonetheless, if you were so inclined and read the docs, you'd see that you can override this default by adding `?$limit=LIMIT` to the API endpoint. For example, to read in only the first five rows, you could use:

```
## Not run
fromJSON("https://data.cityofnewyork.us/resource/nwxe-4ae8.json?$limit=5")
```

Getting back on track, let's plot our tree data just to show it worked. One minor thing I want to point out is that `jsonlite::fromJSON()` automatically coerces everything into a character, so we'll also need to convert some columns to numeric before we plot.

```
nyc_trees %>%
  select(longitude, latitude, stump_diam, spc_common, spc_latin, tree_id) %>%
  mutate(across(longitude:stump_diam, as.numeric)) %>%
  ggplot(aes(x=longitude, y=latitude, size=stump_diam)) +
  geom_point(alpha=0.5) +
  scale_size_continuous(name = "Stump diameter") +
  labs(
    x = "Longitude", y = "Latitude",
    title = "Sample of New York City trees",
    caption = "Source: NYC Open Data"
    )
```



Not too bad. This would probably be more fun / impressive with an actual map of New York behind it. We'll save that for the spatial lecture that's coming up later in the course, though.

Again, I want to remind you that our first application didn't require prior registration on the Open Data NYC website, or creation of an API key. This is atypical. Most API interfaces will only let you access and download data after you have registered an API key with them. This is especially true if you want to access an API linked to a federal agency or institution (Census, BEA, etc.). So let's work through an application where an API key is required…

Application 2: FRED data

Our second application will involve downloading data from the **FRED API**. You will need to register an API key if you want to follow along with my steps, so please do so first before continuing.

As nearly every economist could tell you, FRED is a database maintained by the Federal Reserve Bank of St Louis. You know, the one that let's you plot cool interactive charts like this one of US GNP since 1929.

```
## Sorry, this interactive chart is only available in the the HTML version of the notes.
```

For this second example application, I'm going to show you how to download the data underlying the above chart using the FRED API. In fact, I'll go one better. First I'll show you how to download it yourself, so that you get an understanding of what's happening underneath the hood. Later, I'll direct you to a package that does all of the API work for you.

As with all APIs, a good place to start is the FRED API developer docs. If you read through these, you'd see that the endpoint path we're interested in is **series/observations**. This endpoint "gets the observations or data values for an economic data

series". The endpoint documentation gives a more in-depth discussion, including the various parameters that it accepts.[3] However, the parameters that we'll be focused on here are simply:

- **file_type:** "json" (Not required, but our preferred type of output.)
- **series_id:** "GNPCA" (Required. The data series that we want.)
- **api_key:** "YOUR_API_KEY" (Required. Go and fetch/copy your key now.)

Let's combine these parameters with the endpoint path to view the data directly in our browser. Head over to https://api.stlouisfed.org/fred/series/observations?series_id=GNPCA&api_key=YOUR_API_KEY&file_type=json, replacing "YOUR_API_KEY" with your actual key. You should see something like the following:

At this point you're probably tempted to read the JSON object directly into your R environment using the `jsonlite::readJSON()` function. And this will work. However, that's not what we're going to here. Rather, we're going to go through the **httr** package (link). Why? Well, basically because **httr** comes with a variety of features that allow us to interact more flexibly and securely with web APIs.

Let's start by defining some convenience variables such as the endpoint path and the parameters (which we'll store in a list).

```
endpoint = "series/observations"
params = list(
  api_key= "YOUR_FRED_KEY", ## Change to your own key
  file_type="json",
  series_id="GNPCA"
  )
```

Next, we'll use the `httr::GET()` function to request (i.e. download) the data. I'll assign this to an object called `fred`.

```
# library(httr) ## Already loaded above

fred <-
  httr::GET(
    url = "https://api.stlouisfed.org/", ## Base URL
```

[3] Think of API *parameters* the same way that you think about function *arguments*. They are valid inputs (instructions) that modify the response to an API request.

```
    path = paste0("fred/", endpoint), ## The API endpoint
    query = params ## Our parameter list
    )
```

Take a second to print the `fred` object in your console. What you'll see is pretty cool; i.e. it's the actual API response, including the *Status Code* and *Content*. Something like:

```
## Response [https://api.stlouisfed.org/fred/series/observations?api_key=YOUR_API_KEY&file_type=json&series_id=GN
##   Date: 2019-02-01 00:06
##   Status: 200
##   Content-Type: application/json; charset=UTF-8
##   Size: 9.09 kB
```

To extract the content (i.e. data) from of this response, I'll use the `httr::content()` function. Moreover, since we know that this content is a JSON array, we can convert it to an R object using `jsonlite::fromJSON()` as we did above. With that being said, we *don't* yet know what format the data will taken in R. (SPOILER: Okay, it will be a list.) I could use the base `str()` function to delve into the structure of the object. However, I want to take this opportunity to introduce you to the **listviewer** package (link) ::jsonedit()', which allows for interactive inspection of list objects.[4]

```
fred %>%
  httr::content("text") %>%
  jsonlite::fromJSON() %>%
  listviewer::jsonedit(mode = "view")
```

Luckily, this particular list object isn't too complicated. We can see that we're really interested in the `fred$observations` sub-element. I'll re-run most of the above code and then extract this element. I could do this in several ways, but will use the `purrr::pluck()` function here.

```
fred <-
  fred %>%
  httr::content("text") %>%
  jsonlite::fromJSON() %>%
  purrr::pluck("observations") %>% ## Extract the "$observations" list element
  # .$observations %>% ## I could also have used this
  # magrittr::extract("observations") %>% ## Or this
  as_tibble() ## Just for nice formatting
fred
```

```
## # A tibble: 91 x 4
##    realtime_start realtime_end date       value
##    <chr>          <chr>        <chr>      <chr>
##  1 2020-05-05     2020-05-05   1929-01-01 1120.076
##  2 2020-05-05     2020-05-05   1930-01-01 1025.091
##  3 2020-05-05     2020-05-05   1931-01-01 958.378
##  4 2020-05-05     2020-05-05   1932-01-01 834.291
##  5 2020-05-05     2020-05-05   1933-01-01 823.156
##  6 2020-05-05     2020-05-05   1934-01-01 911.019
##  7 2020-05-05     2020-05-05   1935-01-01 992.537
##  8 2020-05-05     2020-05-05   1936-01-01 1118.944
##  9 2020-05-05     2020-05-05   1937-01-01 1177.572
## 10 2020-05-05     2020-05-05   1938-01-01 1138.989
## #  ... with 81 more rows
```

Okay! We've finally got our data and are nearly ready for some plotting. Recall that `jsonlite::fromJSON()` automatically converts everything to characters, so I'll quickly change some variables to dates (using `lubridate::ymd()`) and numeric.

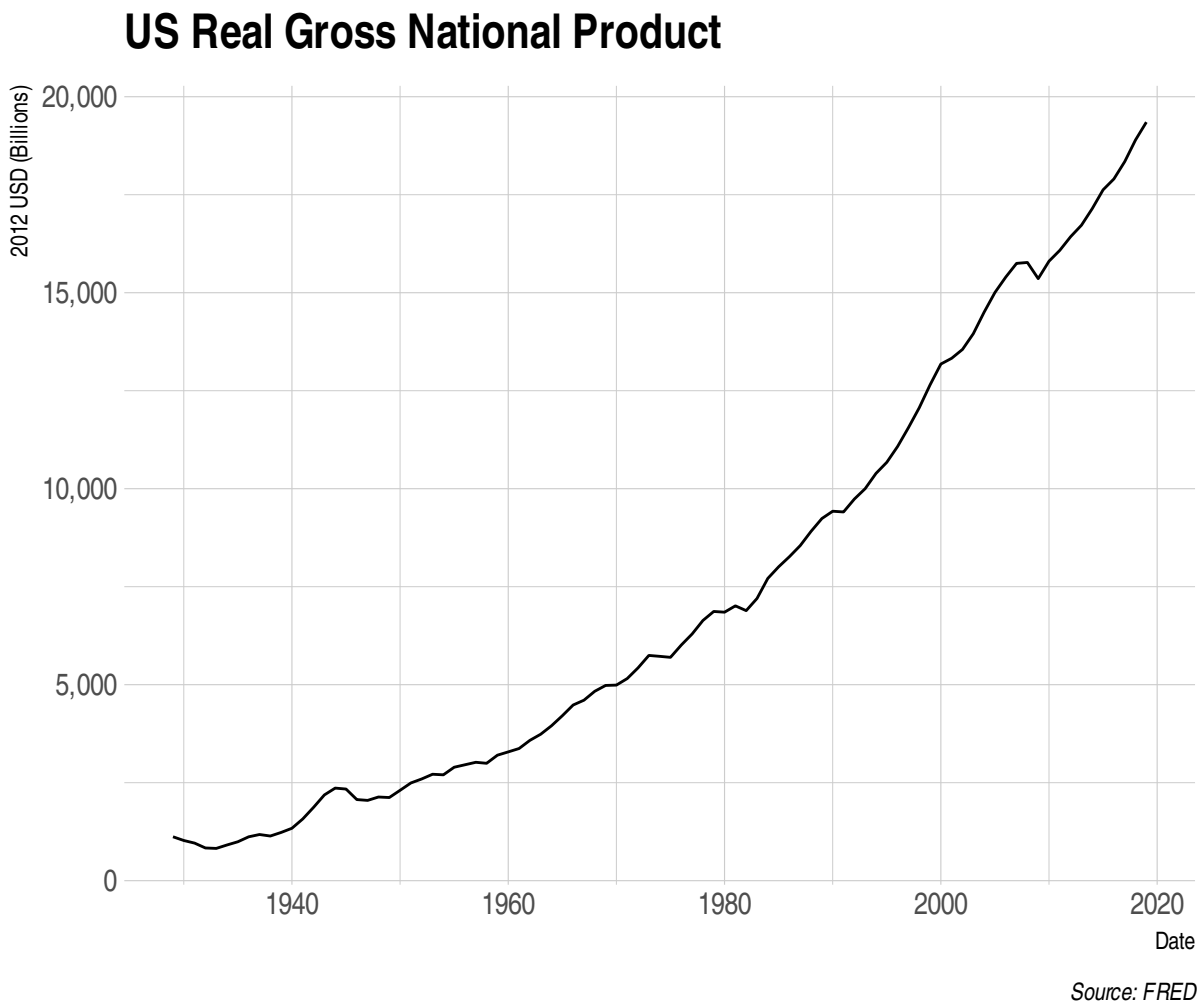
[4] Complex nested lists are the law of the land when it comes to JSON data. Don't worry too much about this now, but the good news is that R ideally suited to parse and handle these nested lists. We'll see more examples later in the course when we start working with programming and spatial data.

```r
# library(lubridate) ## Already loaded above

fred <-
  fred %>%
  mutate(across(realtime_start:date, ymd)) %>%
  mutate(value = as.numeric(value))
```

Let's plot this sucker.

```r
fred %>%
  ggplot(aes(date, value)) +
  geom_line() +
  scale_y_continuous(labels = scales::comma) +
  labs(
    x="Date", y="2012 USD (Billions)",
    title="US Real Gross National Product", caption="Source: FRED"
    )
```



Aside: Safely store and use API keys as environment variables

In the above example, I assumed that you would just replace the "YOUR_FRED_KEY" holder text with your actual API key. This is obviously not very secure or scalable, since it means that you can't share your R script without giving away

your key.[5] Luckily, there's an easy way to safely store and use sensitive information like API keys or passwords: Simply save them as an R **environment variables**. There are two, closely related approaches:

1. Set an environment variable for the current R session only.
2. Set an environment variable that persists across R sessions.

Let's briefly review each of these approaches in turn, followed by a third approach which is even easier (if available).

1) Set an environment variable for the current R session only

Defining an environment variable for the current R session is very straightforward. Simply use the base `Sys.setenv()` function. For example:

```
## Set new environment variable called MY_API_KEY. Current session only.
Sys.setenv(MY_API_KEY="abcdefghijklmnopqrstuvwxyz0123456789")
```

Once this is done, you can then safely assign your key to an object — including within an R Markdown document that you're going to knit and share — using the `Sys.getenv()` function. For example:

```
## Assign the environment variable to an R object
my_api_key <- Sys.getenv("MY_API_KEY")
## Print it out just to show that it worked
my_api_key
```

```
## [1] "abcdefghijklmnopqrstuvwxyz0123456789"
```

Important: While this approach is very simple, note that in practice the `Sys.setenv()` part should only be run directly in your R console. *Never* include code chunks with sensitive `Sys.setenv()` calls in an R Markdown file or other shared documents.[6] That would entirely defeat the purpose! Apart from the annoyance of having to manually set my API key each time I start a new R session, this is one reason that I prefer the next approach of persisting environment variables across sessions…

2) Set an environment variable that persist across R sessions

The trick to setting an R environment variable that is available across sessions is to add it to a special file called `~/.Renviron`. This is a text file that lives on your home directory — note the `~/` path — which R automatically reads upon startup. Because `~/.Renviron` is just a text file, you can edit it with whatever is your preferred text editor. However, you may need to create it first if it doesn't exist. A convenient way to do all of this from RStudio is with the `usethis::edit_r_environ()` function. You will need to run the next few lines interactively:

```
## Open your .Renviron file. Here we can add API keys that persist across R sessions.
usethis::edit_r_environ()
```

This will open up your `~/.Renviron` file in a new RStudio window, which you can then modify as needed. As an example, let's say that you want to add your FRED API key as an environment variable that persists across sessions. You can do this by simply adding a line like the below to your `~/.Renviron` file and saving.[7]

```
FRED_API_KEY="abcdefghijklmnopqrstuvwxyz0123456789" ## Replace with your actual key
```

Once you have saved your changes, you'll need to refresh so that this new environment variable is available in the current session. You could also restart R, but that's overkill.

```
## Optional: Refresh your .Renviron file.
readRenviron("~/.Renviron") ## Only necessary if you are reading in a newly added R environment variable
```

Challenge: Once you've refreshed your `~/.Renviron` file, try to re-download the FRED data from earlier. This time call your FRED API key directly as an environment variable in your parameter list using `Sys.getenv()` like this:

[5] The same is true for compiled R Markdown documents like these lecture notes.

[6] Since the new R environment variable is defined for the duration of the current session, R Markdown will have access to this variable irrespective of whether it was defined in the R Markdown script or not.

[7] I suggest calling it something that's easy to remember like "FRED_API_KEY", but as you wish.

```
params = list(
  api_key= Sys.getenv("FRED_API_KEY"), ## Get API directly and safely from the stored environment variable
  file_type="json",
  series_id="GNPCA"
  )
```

We're going to be revisiting (and setting) environment variables once we get to the cloud computation part of the course. So please make sure that you've understood this section and that your new FRED API key works.

Use a package

One of the great features about the R (and data science community in general) is that someone has probably written a package that does all the heavy API lifting for you. We'll come across many examples during the remainder of this course, but for the moment I want to flag the **fredr** package (link). Take a look at the "Get started" page to see how you could access the same GDP data as above, but this time going through a package.

Application 3: World rugby rankings

Our final application will involve a more challenging case where the API endpoint is *hidden from view*. In particular, I'm going to show you how to access data on **World Rugby rankings**. Because — real talk — what's more important than teaching Americans about rugby?

Note: I used to have a disclaimer here discussing World Rugby's Terms and Conditions for re-use of its data. However, given the HiQ vs. LinkedIn court ruling that we discussed in the previous lecture, that seems redundant now.

Start by taking a look at the complicated structure of the website in a live session. Pay attention to the various tables and other interactive elements like calendars. Now take a minute or two for a quick challenge: Try to scrape the full country rankings using the `rvest` + CSS selectors approach that we practiced last time…

.

.

.

.

.

If you're anything like me, you would have struggled to scrape the desired information using the `rvest` + CSS selectors approach. Even if you managed to extract some kind of information, you're likely only getting a subset of what you wanted. (For example, just the column names, or the first ten rows before the "VIEW MORE RANKINGS" button). And we haven't even considered trying to get information from a different date.[8]

Locating the hidden API endpoint

Fortunately, there's a better way: Access the full database of rankings through the API. First we have to find the endpoint, though. Here's a step-by-step guide of how to that that. It's fairly tedious, but pretty intuitive once you get the hang of it. You can just skip to the GIF below if you would rather see what I did instead of reading through all the steps.

- Start by inspecting the page. (**Ctr**+**Shift**+**I** in Chrome. **Ctrl**+**Shift**+**Q** in Firefox.)
- Head to the **Network** tab at the top of the inspect element panel.
- Click on the **XHR** button.[9]
- Refresh the page (**Ctrl**+**R**). This will allow us to see all the web traffic coming to and from the page in our inspect panel.
- Our task now is to scroll these different traffic links and see which one contains the information that we're after.

[8] Note that the URL doesn't change even when we select a different date on the calendar.
[9] XHR stands for **X**ML**H**ttp**R**equest and is the type of request used to fetch XML or JSON data.

- The top traffic link item references a URL called https://cmsapi.pulselive.com/rugby/rankings/mru?language=en&client=pulse. *Hmmm. "API" you say? "Rankings" you say? Sounds promising...*
- Click on this item and open up the **Preview** tab.
- In this case, we can see what looks to be the first row of the rankings table ("New Zealand", etc.)
- To make sure, you can grab that https://cmsapi.pulselive.com/rugby/rankings/mru?language=en&client=pulse, and paste it into our browser (using the JSONView plugin) from before.

Sweet. Looks like we've located our API endpoint. As promised, here's GIF of me walking through these steps in my browser:

```
## Sorry, this GIF only available in the the HTML version of the notes.
```

Pulling the data into R

Let's pull the data from the API endpoint into R. Again, I'll be using `jsonlite::readJSON()` function.

```
endpoint <- "https://cmsapi.pulselive.com/rugby/rankings/mru?language=en&client=pulse"
rugby <- fromJSON(endpoint)
str(rugby)
```

```
## List of 3
##  $ label    : chr "Mens Rugby Union"
##  $ entries  :'data.frame':   105 obs. of  6 variables:
##   ..$ team       :'data.frame':  105 obs. of  5 variables:
##   .. ..$ id          : int [1:105] 39 37 34 36 42 33 38 35 49 40 ...
##   .. ..$ altId       : logi [1:105] NA NA NA NA NA NA ...
##   .. ..$ name        : chr [1:105] "South Africa" "New Zealand" "England" "Ireland" ...
##   .. ..$ abbreviation: chr [1:105] "RSA" "NZL" "ENG" "IRE" ...
##   .. ..$ annotations : logi [1:105] NA NA NA NA NA NA ...
##   ..$ matches    : int [1:105] 212 220 208 199 203 216 227 194 174 190 ...
##   ..$ pts        : num [1:105] 94.2 92.1 88.4 84.9 82.7 ...
##   ..$ pos        : int [1:105] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ previousPts: num [1:105] 94.2 92.1 88.3 84.9 83.9 ...
##   ..$ previousPos: int [1:105] 1 2 3 4 5 6 7 8 9 10 ...
##  $ effective:List of 3
##   ..$ millis   : num 1.58e+12
##   ..$ gmtOffset: num 0
##   ..$ label    : chr "2020-03-09"
```

We have a nested list, where what looks to be the main element of interest, `$entries`, is itself a list.[10] Let's take a closer look at the `$entries` element. We could use the `str()` function again to do so. But `listviewer::jsonedit()` is ideally suited for exploring nested lists in an interactive session, so that's what I recommend you use.

```
# str(rugby$entries) ## Base option
listviewer::jsonedit(rugby, mode = "view")
```

For completeness, let's take a peak at the `rugby$entries$team` data frame to confirm that it has information that is useful to us.

```
head(rugby$entries$team)
```

```
##   id altId         name abbreviation annotations
## 1 39    NA South Africa          RSA          NA
## 2 37    NA  New Zealand          NZL          NA
## 3 34    NA      England          ENG          NA
## 4 36    NA      Ireland          IRE          NA
```

[10] I know that R says `rugby$entries` is a data.frame, but we can tell from the `str()` call that it follows a list structure. In particular, the `rugby$entries$team` sub-element is a itself data frame.

```
## 5 42    NA       France          FRA          NA
## 6 33    NA        Wales          WAL          NA
```

Okay, a clearer picture is starting to emerge. It looks like we can just bind the columns of the `rugby$entries$team` data frame directly to the other elements of the parent `$entries` "data frame" (actually: "list"). Let's do that using `dplyr::bind_cols()` and then clean things up a bit. I'm going to call the resulting data frame `rankings`.

```
# library(janitor) ## Already loaded above

rankings <-
  bind_cols(
    rugby$entries$team,
    rugby$entries %>% select(matches:previousPos)
    ) %>%
  clean_names() %>%
  select(-c(id, alt_id, annotations)) %>% ## These columns aren't adding much of interest
  select(pos, pts, everything()) %>% ## Reorder remaining columns
  as_tibble() ## "Enhanced" tidyverse version of a data frame
rankings
```

```
## # A tibble: 105 x 7
##      pos   pts name         abbreviation matches previous_pts previous_pos
##    <int> <dbl> <chr>        <chr>          <int>        <dbl>        <int>
##  1     1  94.2 South Africa RSA              212         94.2            1
##  2     2  92.1 New Zealand  NZL              220         92.1            2
##  3     3  88.4 England      ENG              208         88.3            3
##  4     4  84.9 Ireland      IRE              199         84.9            4
##  5     5  82.7 France       FRA              203         83.9            5
##  6     6  82.6 Wales        WAL              216         82.8            6
##  7     7  81.9 Australia    AUS              227         81.9            7
##  8     8  80.7 Scotland     SCO              194         79.6            8
##  9     9  79.3 Japan        JPN              174         79.3            9
## 10    10  78.3 Argentina    ARG              190         78.3           10
## # ... with 95 more rows
```

BONUS: Get and plot the rankings history

NOTE: This bonus section involves some programming and loops. I know that we haven't gotten to the programming section of the course, so don't worry about the specifics of the next few code chunks. I'll try to comment my code quite explicitly, but I mostly want you to focus on the big picture.

The above table looks great, except for the fact that it's just a single snapshot of the most recent rankings. We are probably more interested in looking back at changes in the ratings over time.

But how to do this? Well, in the spirit of art-vs-science, let's open up the Inspect window of the rankings page again and start exploring. What happens if we click on the calendar element, say, change the year to "2018" and month to "May"? (Do this yourself.)

This looks promising! Essentially, we get the same API endpoint that we saw previously, but now appended with a date, https://cmsapi.pulselive.com/rugby/rankings/mru?date=2018-05-01&client=pulse. If you were to continue along in this manner — clicking on the website calendar and looking for XHR traffic — you would soon realise that these date suffixes follow a predictable pattern: They are spaced out a week apart and always fall on a Monday. In other words, World Rugby updates its international rankings table weekly and publishes the results on Mondays.

We now have enough information to write a function that will loop over a set of dates and pull data from the relevant API endpoint. To start, we need a vector of valid dates to loop over. I'm going to use various functions from the `lubridate` package to help with this. Note that I'm only to extract a few data points — one observation a year for the last decade or so — since I only want to demonstrate the principle. No need to hammer the host server. (More on that below.)

```
## We'll look at rankings around Jan 1st each year. I'll use 2004 as an
## arbitrary start year and then proceed until the present year.
start_date <- ymd("2004-01-01")
end_date <- floor_date(today(), unit="years")
dates <- seq(start_date, end_date, by="years")
## Get the nearest Monday to Jan 1st to coincide with rankings release dates.
dates <- floor_date(dates, "week", week_start = getOption("lubridate.week.start", 1))
dates
```

```
##  [1] "2003-12-29" "2004-12-27" "2005-12-26" "2007-01-01" "2007-12-31"
##  [6] "2008-12-29" "2009-12-28" "2010-12-27" "2011-12-26" "2012-12-31"
## [11] "2013-12-30" "2014-12-29" "2015-12-28" "2016-12-26" "2018-01-01"
## [16] "2018-12-31" "2019-12-30"
```

Next, I'll write out a function that I'll call `rugby_scrape`. This function will take a single argument: a date that it will use to construct a new API endpoint during each iteration. Beyond that, it will pretty do much exactly the same things that we did in our previous, manual data scrape. The only other difference is that it will wait three seconds after running (i.e. `Sys.sleep(3)`). I'm adding this final line to avoid hammering the server with instantaneous requests when we put everything into a loop.

```
## First remove our existing variables. This is not really necessary, since R is smart enough
## to distinguish named objects in functions from named objects in our global environment.
## But I want to emphasise that we're creating new data here and avoid any confusion.
rm(rugby, rankings, endpoint)

## Now, create the function. I'll call it "rugby_scrape".
rugby_scrape <-
  function(x) {
    endpoint <- paste0("https://cmsapi.pulselive.com/rugby/rankings/mru?date=", x, "&client=pulse")
    rugby <- fromJSON(endpoint)
    rankings <-
      bind_cols(
        rugby$entries$team,
        rugby$entries %>% select(matches:previousPos)
        ) %>%
      clean_names() %>%
      mutate(date = x) %>% ## New column to keep track of the date
      select(-c(id, alt_id, annotations)) %>% ## These columns aren't adding much of interest
      select(date, pos, pts, everything()) %>% ## Reorder remaining columns
      as_tibble() ## "Enhanced" tidyverse version of a data frame
    Sys.sleep(3) ## Be nice!
    return(rankings)
  }
```

Finally, we can now iterate (i.e. loop) over our `dates` vector, by plugging the values sequentially into our `rugby_scrape` function. There are a variety of ways to iterate in R, but I'm going to use an `lapply()` call below.[11] We'll then bind everything into a single data frame using `dplyr::bind_rows()` and name the resulting object `rankings_history`.

```
rankings_history <-
  lapply(dates, rugby_scrape) %>% ## Run the iteration
  bind_rows() ## Bind the resulting list of data frames into a single data frame
rankings_history
```

```
## # A tibble: 1,674 x 8
##    date         pos   pts name    abbreviation matches previous_pts previous_pos
```

[11] Again, don't worry too much about this now. We'll cover iteration and programming in more depth in a later lecture.

```
##    <date>     <int> <dbl> <chr>   <chr>          <int>        <dbl>        <int>
##  1 2003-12-29     1  94.0 England ENG              17         92.1            1
##  2 2003-12-29     2  90.1 New Ze~ NZL              17         88.2            3
##  3 2003-12-29     3  86.6 Austra~ AUS              17         88.4            2
##  4 2003-12-29     4  82.7 France  FRA              17         84.7            4
##  5 2003-12-29     5  81.2 South ~ RSA              15         81.2            5
##  6 2003-12-29     6  80.5 Ireland IRE              15         80.5            6
##  7 2003-12-29     7  78.0 Argent~ ARG              14         78.0            7
##  8 2003-12-29     8  76.9 Wales   WAL              15         76.9            8
##  9 2003-12-29     9  76.4 Scotla~ SCO              15         76.4            9
## 10 2003-12-29    10  73.5 Samoa   SAM              14         73.5           10
## # ... with 1,664 more rows
```

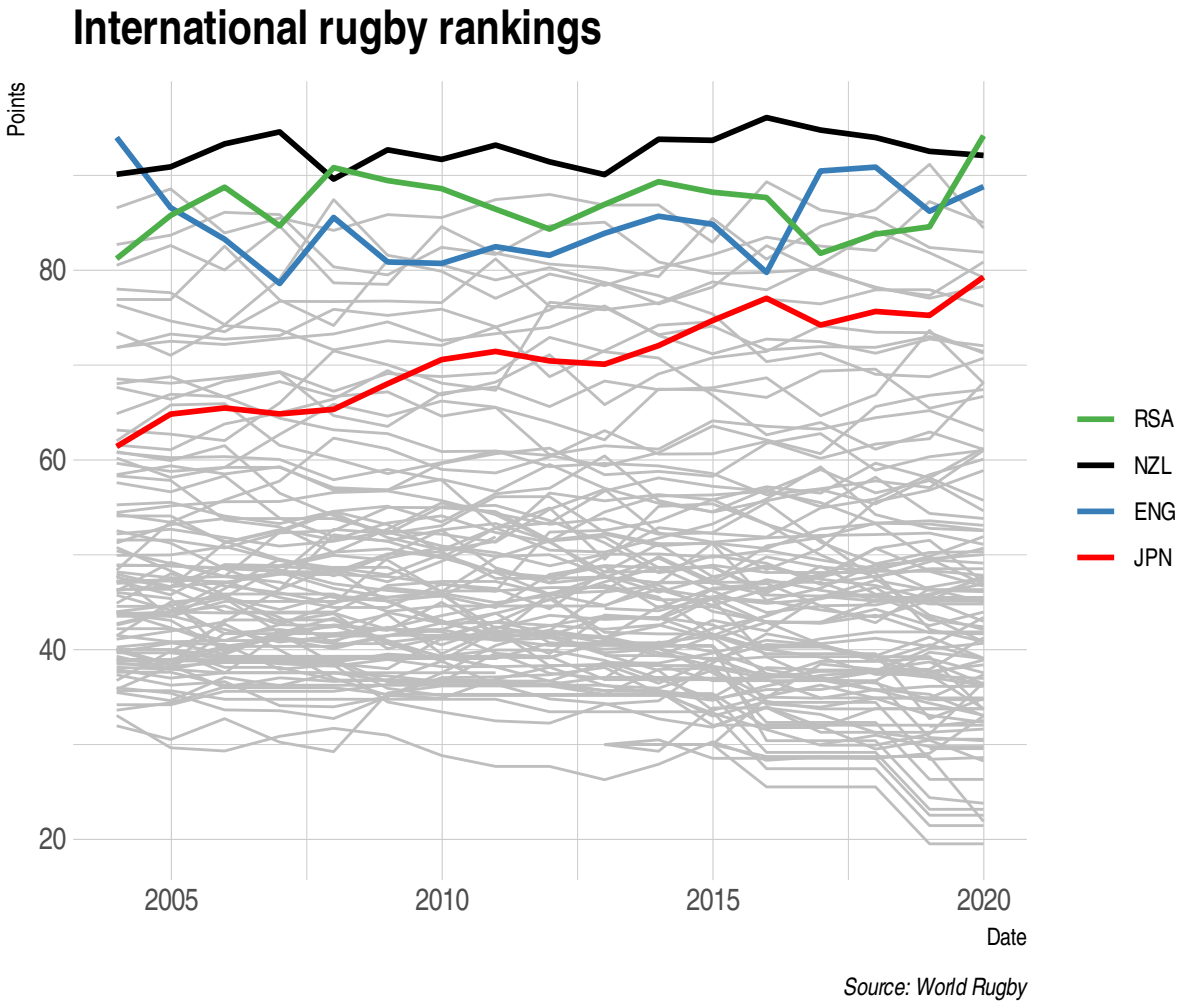
Let's review what we just did:

- We created a vector of dates — creatively called `dates` — with observations evenly spaced (about) a year apart, falling on the Monday closest to Jan 1st for that year.
- We then iterated (i.e. looped) over these dates using a function, `rugby_scrape`, which downloaded and cleaned data from the relevant API endpoint.
- At the end of each iteration, we told R to wait a few seconds before executing the next step. Remember that R can execute these steps much, much quicker than we could ever type them manually. It probably doesn't matter for this example, but you can easily "overwhelm" a host server by hammering it with a loop of automated requests. (Or, just as likely: They have safeguards against this type of behaviour and will start denying your requests as a suspected malicious attack.) As ever, the "be nice" motto holds sway when scraping web data.
- Note that each run of our iteration will have produced a separate data frame, which `lapply()` by default appends into a list. We used `dplyr::bind_rows()` to bid these separate data frames into a single data frame.

Okay! Let's plot the data and highlight a select few countries in the process.

```
teams <- c("NZL", "RSA", "ENG", "JPN")
team_cols <- c("NZL"="black", "RSA"="#4DAF4A", "ENG"="#377EB8", "JPN" = "red")

rankings_history %>%
  ggplot(aes(x=date, y=pts, group=abbreviation)) +
  geom_line(col = "grey") +
  geom_line(
    data = rankings_history %>% filter(abbreviation %in% teams),
    aes(col=fct_reorder2(abbreviation, date, pts)),
    lwd = 1
    ) +
  scale_color_manual(values = team_cols) +
  labs(
    x = "Date", y = "Points",
    title = "International rugby rankings", caption = "Source: World Rugby"
    ) +
  theme(legend.title = element_blank())
```



New Zealand's extended dominance in the global game is extraordinary, especially given its tiny population size. They truly do have a legitimate claim to being the greatest international team in the history of professional sport.[12] OTOH, South African rugby supporters can finally (finally!) rejoice after a long dry spell. Bring 'er home, Siya, bring 'er home.

Summary

- An API is a set of rules and methods that allow one computer or program (e.g. host server) to talk to another (e.g. client or browser).
- We can access information through an API directly by specifying a valid API endpoint.
  - The API endpoint for most web-based applications will be a URL with either JSON or XML content.
- Some APIs don't require an access key or token, but most do. You can add this key as a parameter to the API endpoint.
- Downloading content from an API endpoint to our local computer (i.e. R environment) can be done in a variety of ways.
  - E.g. `jsonlite::readJSON()` to read the the JSON array directly, or `httr::GET()` to download the entire response, or installing a package that does the job for us.
- Next lecture: We start of the programming section of the course.

Further resources and exercises

- Tyler Clavelle has written several cool blog posts on interacting with APIs through R. I especially recommend going

[12] Obligatory link to the best ever haka.

over — and replicating — his excellent tutorial on the GitHub API.

- Jonathan Regenstein has a nice post on RStudio's *R Views* blog, "GDP Data via API", which treads a similar path to my FRED example. Except he uses the Bureau of Economic Analysis (BEA) API.
- Greg Reda's "Web Scraping 201: finding the API" covers much of the same ground as we have here. While he focuses on Python tools, I've found it to be a handy reference over the years. (You can also take a look at the earlier posts in Greg's webscraping series — Part 1 and Part 2 — to see some Python equivalents of the `rvest` tools that we've been using.)
- Ian London (another Python user) has a nice blog post on "Discovering hidden APIs" from Airbnb.