10: Data Scraping

Environmental Data Analytics | John Fay

Fall 2024

Objectives

1. Acquire and scrape data from web sources
2. Process web-scraped data into reproducible formats
3. Use wrangling techniques to automate scraping processes

Set up

```
library(tidyverse)
library(lubridate)
library(here); here()
```

```
## [1] "/home/guest/EDE_Fall2024"
```

```
#install.packages("rvest")
library(rvest)

# Set theme
mytheme <- theme_classic() +
  theme(axis.text = element_text(color = "black"),
        legend.position = "top")
theme_set(mytheme)
```

Part 1. Scraping data

What does it mean to "scrape data" from the web?

The internet is a **vast** source of data. Often websites will share links to download data directly. And more and more organizations are providing application programming interfaces or APIs that allow you to tap in to on-line datasets programmatically. Still, some data that may be useful to you might appear on a website, but the website doesn't offer an easy way of grabbing it into your coding environment. It's this last example where scraping comes in handy.

Web scraping is the process of pulling data directly from websites. Copy and paste is a brute force form of web scraping, but one that often also involves a lot of clean up of your data. A slightly more elegant, yet still nimble approach is to navigate the code in which the web page was written and perhaps glean the data you want from there. This doesn't always work, and it still takes some trial and error, but it can give you access to invaluable datasets in a pinch.

Some important caveats

You might now feel that web scraping isn't always legitimate - and you're right! Not all content providers want their sites to be scraped. To ensure that your web scraping is done ethically, check out this great article covering the ethics and legality of web scraping: https://prowebscraper.com/blog/is-web-scraping-legal/

> For discussion: Can you give examples of ethical and unethical web scraping?

1.1 Exploring the data we want to scrape

- Navigate to https://www.ncwater.org/WUDC/app/WWATR/report. This site lists water use reports submitted by various industries around NC.
- Click on the link to view the report for "Plant 15" facility in 2020: https://www.ncwater.org/WUDC/app/WWATR/report/view/0004-0001/2020
- View the data we can scrape on this page.

What we want to do first is scrape the average daily water withdrawal for a given site along with the site's "registrant" and their "facility type". Once we see how it's done with one site, we'll look at how we can iterate through many sites and grab the data we want data from each.

1.2 Fetching the contents

To begin, we'll fetch the contents of the web site into our coding environment using RVest's `read_html()` function.

```
#Fetch the web resources from the URL
webpage <- read_html('https://www.ncwater.org/WUDC/app/WWATR/report/view/0004-0001/2020')
webpage
```

```
## {html_document}
## <html xmlns="http://www.w3.org/1999/xhtml">
## [1] <head>\n<title>DWR :: Water Withdrawal and Transfer Registration</title>\ ...
## [2] <body>\r\n\r\n    <div id="header">\r\n    <h1 id="site-title">\n<span>NC ...
```

Understanding web pages from a coding standpoint

The `read_html()` function returns a list of the two elements,

and

, that comprise a typical web page. * The

mostly contains information and instructions on how the page should be constructed. * The

mostly contains the content that is displayed in your browser.

EXERCISE: Open the source of the web page in your browser (-). * Do you see the

and

sections? * Do you see where it states the registrant? ("American & Efird, Inc.")

HTML files follow a syntax of **tags** (what falls beteween the < and >) and **values**. Tags can indicate what kind of hyper-text element it is, and they can also include **properties** such as "id" and "class" to which specific **property values** can be assigned. This is important to know at some level, because it's this system

of tags, properties, and property values that allow us to target specific elements in a web page so that we can scrape the data we want. You'll also see that items are hierarchical with some items (e.g.

containing other items

).

1.3 Scraping the data

Now we want to scrape some data shown in the page into our R coding environment. The bits we want to scrape are: * The name of the Registrant: "American & Efird, Inc." * The name of the facility: "Plant 15" * The facility type: "Industrial" * And the average water withdrawals for each month: "0.584. 0.647, 0.543, etc."

To do this we first need to install a tool on our web browser to be able to call the web text we need. The tool is called a Selector Gadget, which for Chrome can be found here.

The selector gadget tool is useful in identifying the internal ids of the elements shown in the web page that we want to extract. Using it is a bit clumsy, often requiring a bit of mucking about, but it still makes our work a lot easier. We'll begin by using it to determine the id of the box containing the registrant of our our site. 1. Activate the Selector Gadget tool. 2. Click on the box listing the registrant ("American & Efird, Inc."). You'll see a number of boxes in yellow, but we just want the one box in yellow. 3. Click various other boxes until only the one we want is highlighted in yellow (or green). It's ok if others are in red. This may take a bit of trial and error, and if you get too mess up, you can start over by clicking the `Clear` button in the Selector Gadget floating toolbar.

When successful, the box should display `.table tr:nth-child(1) td:nth-child(2)`. This uniquely identifies the html element containing the data we want.

With this id known, we can extract it's value in a two-step process. First we use `html_nodes()` to grab the element using its tag. Then we extract the value associated with that element text using the `html_text()` function.

```
the_registrant <- webpage %>%
  html_nodes(".table tr:nth-child(1) td:nth-child(2)") %>%
  html_text()
the_registrant
```

```
## [1] "American & Efird, Inc."
```

EXERCISE: In the code chunk below, see if you can extract the Facility Name and Facility Type from the web page. >The links you should receive from the Selector Gadget are - "tr:nth-child(2) th+ .left:nth-child(2)" - "tr:nth-child(2) .left~ .left+ td.left"

```
the_facility_name <- webpage %>%
  html_nodes("tr:nth-child(2) th+ .left:nth-child(2)") %>%
  html_text()
the_facility_name
```

```
## [1] "Plant 15"
```

```
the_facility_type <- webpage %>%
 html_nodes("tr:nth-child(2) .left~ .left+ td.left") %>%
  html_text()
the_facility_type
```

```
## [1] "Industrial"
```

Next, we'll extract multiple items from the web page into a list object. Using Selector Gadget, highlight in yellow or green all the Average Daily Withdrawal (MGD) values from the web page. Tip: Hold the key down while clicking elements to add/subtract regions from selected regions.

Below is the code to extract average daily withdrawls. Add your code to extract max daily withdrawals into the `max_withdrawals` variable.

```
avg_withdrawals <- webpage %>%
  html_nodes('.table:nth-child(7) td:nth-child(7) , .table:nth-child(7) td:nth-child(3)') %>%
  html_text()
avg_withdrawals
```

```
##  [1] ".584" ".492" ".647" ".613" ".543" ".683" ".371" ".650" ".349" ".535"
## [11] ".447" ".449"
```

```
max_withdrawals <- webpage %>%
  html_nodes('.table:nth-child(7) td:nth-child(8) , .table:nth-child(7) td:nth-child(4)') %>%
  html_text()
max_withdrawals
```

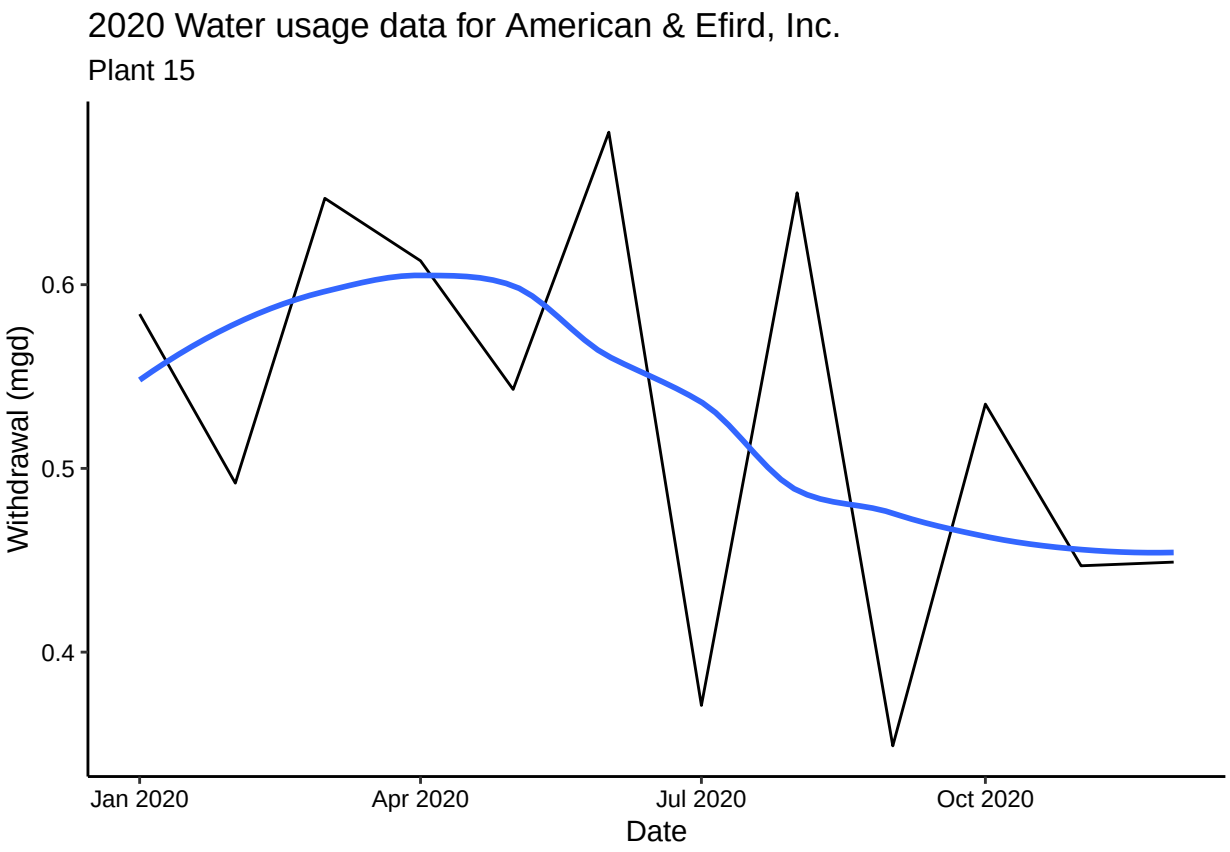
```
##  [1] "1.029" "1.045" "1.056" "1.115" ".934"  "1.096" ".978"  "1.018" "1.146"
## [10] "1.046" "1.111" ".889"
```

1.4 Construct a dataframe from the data & plot the values

Now that we've scraped the data we want into our coding environment, the next step is to wrangle these data into formats we can work with more easily, i.e. a dataframe. Here, we emply the techniques we've learned earlier in this class to do this and then plot the data.

```
#Create a dataframe of withdrawals
df_withdrawals <- data.frame("Month" = rep(1:12),
                             "Year" = rep(2020,12),
                             "Avg_Withdrawals_mgd" = as.numeric(avg_withdrawals),
                             "Max_Withdrawals_mgd" = as.numeric(max_withdrawals))

#Modify the dataframe to include the facility name and type as well as the date (as date object)
df_withdrawals <- df_withdrawals %>%
  mutate(Registrant = !!the_registrant,
         Facility_name = !!the_facility_name,
         Facility_type = !!the_facility_type,
         Date = my(paste(Month,"-",Year)))

#Plot
ggplot(df_withdrawals,aes(x=Date,y=Avg_Withdrawals_mgd)) +
  geom_line() +
  geom_smooth(method="loess",se=FALSE) +
  labs(title = paste("2020 Water usage data for",the_registrant),
       subtitle = the_facility_name,
       y="Withdrawal (mgd)",
       x="Date")
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



Part 2. Automating the scraping process

2.1 Streamline the process

Data! But let's keep going. Next, we'll streamline what we've learned above in an effort to automate the process.

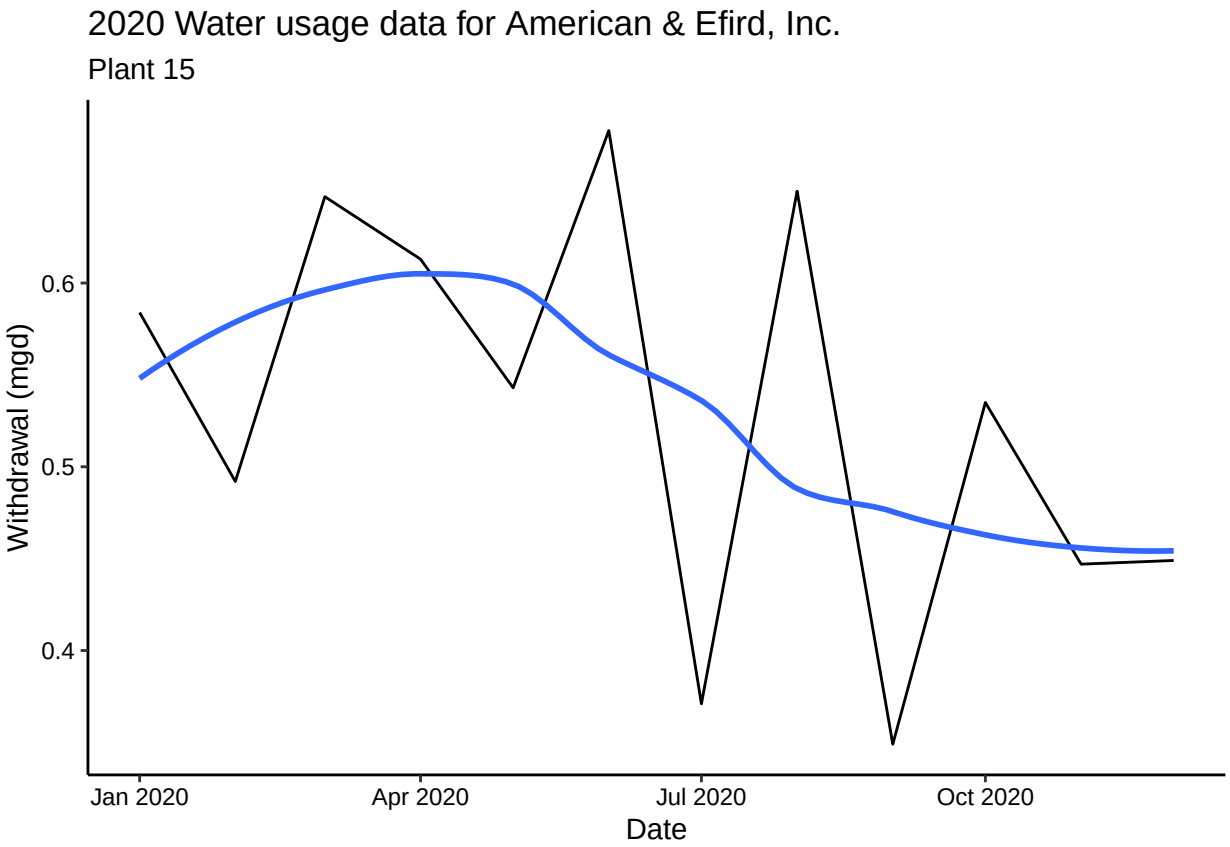
This begins by setting the parameters of the page we'll be scraping to variables so we can more easily point to, say, a different facility or a different year. Then we update our other code to use these parameters to scrape the data.

```
#Construct the scraping web address, i.e. its URL
the_base_url <- 'https://www.ncwater.org/WUDC/app/WWATR/report/view'
the_facility <- '0004-0001'
the_year <- 2020
the_scrape_url <- paste0(the_base_url, '/', the_facility, '/', the_year)
print(the_scrape_url)
```

```
## [1] "https://www.ncwater.org/WUDC/app/WWATR/report/view/0004-0001/2020"
```

```r
#Retrieve the website contents
the_website <- read_html(the_scrape_url)

#Set the element address variables (determined in the previous step)
the_registrant_tag <- '.table tr:nth-child(1) td:nth-child(2)'
the_facility_name_tag <- 'tr:nth-child(2) th+ .left:nth-child(2)'
the_facility_id_tag <- 'tr:nth-child(2) .left~ .left+ td.left'
the_data_tag <- '.table:nth-child(7) td:nth-child(7) , .table:nth-child(7) td:nth-child(3)'

#Scrape the data items
the_registrant <- the_website %>% html_nodes(the_registrant_tag) %>% html_text()
the_facility_name <- the_website %>%   html_nodes(the_facility_name_tag) %>%  html_text()
the_facility_type <- the_website %>% html_nodes(the_facility_id_tag) %>% html_text()
the_withdrawals <- the_website %>% html_nodes(the_data_tag) %>% html_text()

#Construct a dataframe from the scraped data
df_withdrawals <- data.frame("Month" = rep(1:12),
                             "Year" = rep(the_year,12),
                             "Avg_Withdrawals_mgd" = as.numeric(the_withdrawals)) %>%
  mutate(Registrant = !!the_registrant,
         Facility_name = !!the_facility_name,
         Facility_type = !!the_facility_type,
         Date = my(paste(Month,"-",Year)))

#Plot
ggplot(df_withdrawals,aes(x=Date,y=Avg_Withdrawals_mgd)) +
  geom_line() +
  geom_smooth(method="loess",se=FALSE) +
  labs(title = paste("2020 Water usage data for",the_registrant),
       subtitle = the_facility_name,
       y="Withdrawal (mgd)",
       x="Date")
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



Run the above; it should produce the same result as the previous R chunk. HOWEVER, change the year variable to `2015` and re-run the chunk. Change the facility ID to `0218-0238` and run again. Now, we have a nifty little scraping tool!

2.2 Automation, Step 1: Build a function

We have our code so we can fairly easily scrape any site (if we know its ID) for any year. Let's improve our code so we can automate the process more easily and perhaps scrape many years worth of data. To make this process run more easily, we'll first convert our code into a function that produces a dataframe of withdrawal data for a given year and facility ID.

```
#Create our scraping function
scrape.it <- function(the_year, the_facility){

  #Retrieve the website contents
  the_website <- read_html(paste0('https://www.ncwater.org/WUDC/app/WWATR/report/view/',
                                  the_facility, '/', the_year))

  #Set the element address variables (determined in the previous step)
  the_registrant_tag <- '.table tr:nth-child(1) td:nth-child(2)'
  the_facility_name_tag <- 'tr:nth-child(2) th+ .left:nth-child(2)'
  the_facility_id_tag <- 'tr:nth-child(2) .left~ .left+ td.left'
  the_data_tag <- '.table:nth-child(7) td:nth-child(7) , .table:nth-child(7) td:nth-child(3)'

  #Scrape the data items
```

```
  the_registrant <- the_website %>% html_nodes(the_registrant_tag) %>% html_text()
  the_facility_name <- the_website %>%  html_nodes(the_facility_name_tag) %>%  html_text()
  the_facility_type <- the_website %>% html_nodes(the_facility_id_tag) %>% html_text()
  avg_withdrawals <- the_website %>% html_nodes(the_data_tag) %>% html_text()

  #Convert to a dataframe
  df_withdrawals <- data.frame("Month" = rep(1:12),
                               "Year" = rep(the_year,12),
                               "Avg_Withdrawals_mgd" = as.numeric(avg_withdrawals)) %>%
    mutate(Registrant = !!the_registrant,
           Facility_name = !!the_facility_name,
           Facility_type = !!the_facility_type,
           Date = my(paste(Month,"-",Year)))

  #Pause for a moment - scraping etiquette
  #Sys.sleep(1) #uncomment this if you are doing bulk scraping!

  #Return the dataframe
  return(df_withdrawals)
}
```

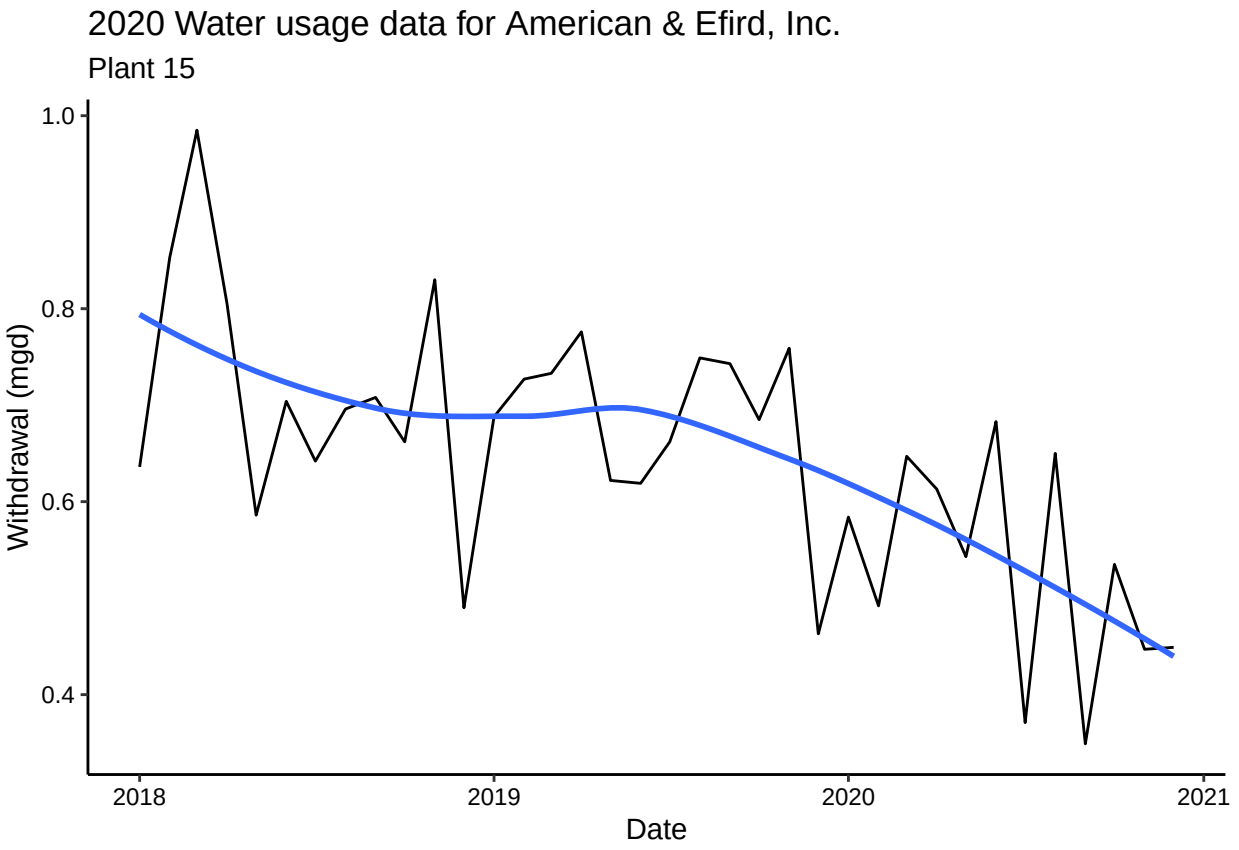
```
#Run the function
the_df <- scrape.it(2018,'0004-0001')
view(the_df)
```

2.3 Use iterators to retrieve data for a set of years

R can iterate using either "lapply" or Purrrs' "map" functions.

```
#Set the inputs to scrape years 2015 to 2020 for the site "0004-0001"
the_years = rep(2018:2020)
my_facility = '0004-0001'

#Use lapply to apply the scrape function
the_dfs <- lapply(X = the_years,
                  FUN = scrape.it,
                  the_facility=my_facility)

#OR, use purrr's map function
the_dfs <- map(the_years,scrape.it,the_facility=my_facility)

#Conflate the returned dataframes into a single dataframe
the_df <- bind_rows(the_dfs)

#Plot, because it's fun and rewarding
ggplot(the_df,aes(x=Date,y=Avg_Withdrawals_mgd)) +
  geom_line() +
  geom_smooth(method="loess",se=FALSE) +
  labs(title = paste("2020 Water usage data for",the_registrant),
       subtitle = the_facility_name,
       y="Withdrawal (mgd)",
       x="Date")
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



Part 3. Web Crawling

3.1 Get a list of facility id's via scraping

Web crawling refers to navigating within a web site to scrape data across multiple sub-pages. In the exercise above, data are scraped from just one site. but what if we wanted to grab data from all sites? To do that, we'd need to drill into each site listed on the main page.

To do this just takes just a bit of more advanced scripting and scraping. Here we show how you can scrape multiple tags into a list and then "crawl" into each site and grab its data.

```
#Set the web page
the_main_url <- "https://www.ncwater.org/WUDC/app/WWATR/report"

#Pull its contents into our code environment
the_main_website <- read_html(the_main_url)

#Use Selector Gadget to find the tag for all the "View Report" objects
#You should find this to be "#content a"

#Then extract all the nodes associated with this and rather than extracting them
the_facility_ids <- the_main_website %>%
```

```
  html_nodes('#content a') %>%    #Extract a list of html nodes taged with "content a"
  html_attr("href") %>%           #Pull the "href" attribute from the node
  str_split("/") %>%              #Split the html address into substrings, separated by "/"
  map_chr(.,9)                    #Extract the 9th item in the above, which is the facility ID

#Have a look:
head(the_facility_ids)
```

```
## [1] "0831-0001" "0847-0001" "0043-0001" "0876-0001" "0004-0001" "0141-0001"
```

3.2 Scrape 2020 water usage data for all sites

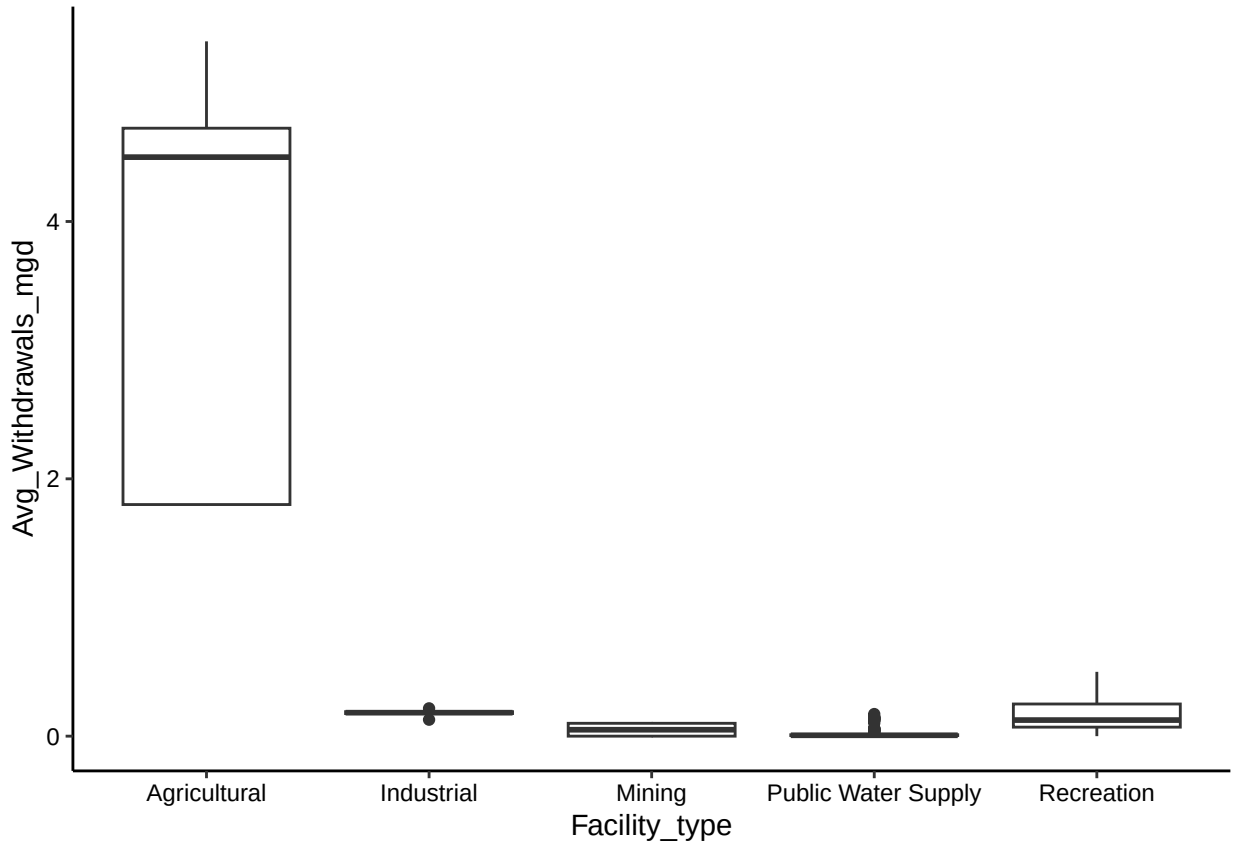
We have our "scrape.it" function and our list of facility IDs. We can let the code do the work and scrape all the data for 2020. Here, to limit our demand on the DWR server, we'll just do a subset of facilities.

We use Purrr's `map2` function which allow us to provide two vectors of equal length as parallel inputs such that the function provided (our `scrape.it` function) is run with element 1 of each list, then element 2, and so forth. This is why we need to create a list of the one year we want that is the same length as the sample of facility ids.

This website gives a nice example of this: https://dcl-prog.stanford.edu/purrr-parallel.html

```
#Subset the facilities
the_facility_ids_subset <-  sample(the_facility_ids,20)

#Create a list of the year we want, the same length as the vector above
the_years <- rep.int(2020,length(the_facility_ids_subset))

#"Map" the "scrape.it" function to retrieve data for all these
dfs_2020 <- map2(the_years, the_facility_ids_subset, scrape.it)

#Conflate the returned list of dataframes into a single one
df_2020 <- bind_rows(dfs_2020)


#Plot
ggplot(df_2020,aes(y = Avg_Withdrawals_mgd, x=Facility_type)) +
  geom_boxplot()
```

```
## Warning: Removed 43 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```



3.3 Scraping across years and sites.

To scrape for both years and sites, we need to dig deeper into Purrr's map functions. Most importantly we need to use the `cross2()` function to create a list of each unique combination of `year` and `facility_id`. We then use the `lift()` function to extract these elements in a way that can be easily mapped to our function.

```
#Create a subset of facilities
the_facilities <- sample(the_facility_ids,3)
the_years = c(2018,2019,2020)

#Use the cross2 function to generate a list of unique combos of year and facility ids
#Then send apply our scrape.it function to grab values into a list of dataframes
the_df <- cross2(the_years,the_facilities) %>%
  map(lift(scrape.it)) %>%
  bind_rows()
```

```
## Warning: `cross2()` was deprecated in purrr 1.0.0.
## i Please use `tidyr::expand_grid()` instead.
## i See <https://github.com/tidyverse/purrr/issues/768>.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.

## Warning: `lift()` was deprecated in purrr 1.0.0.
```

```
## This warning is displayed once every 8 hours.
## Call ‘lifecycle::last_lifecycle_warnings()‘ to see where this warning was
## generated.
```

```
#Plot
ggplot(the_df,aes(x=Date,y=Avg_Withdrawals_mgd,color=Facility_name)) +
  #geom_line() +
  geom_smooth(method="loess",se=FALSE) +
  labs(title = "Water usage data",
       y="Withdrawal (mgd)",
       x="Date")
```

```
## ‘geom_smooth()‘ using formula = ’y ~ x’
```