Retrieving Data From The Web

Ista Zahn

Workshop overview and materials

Workshop description

Data is more plentiful and in many ways easier to acquire than ever. That doesn't always mean a perfectly cleaned and organized data set is waiting for you to download it! More commonly data needs to be retrieved from a web service or web page and assembled into a usable data set. This workshop will show you how to do that, using the httr and rvest packages in R.

Prerequisites and Preparation

Prior to the workshop you should:

- install R from https://cran.r-project.org/
- install RStudio from https://www.rstudio.com/products/rstudio/download/#download
- install the tidyverse package in R with `install.packages("tidyverse")`
- download and extract the workshop materials from https://github.com/izahn/R_retrieving_data/archive/retrieving_data_v1.zip

During the workshop you must have a laptop with the above software installed and a working internet connection.

The lesson notes are included in the download link above. You can also view the lesson notes at https://rawgit.com/izahn/R_retrieving_data/master/retrievingOnlineData.html

A github repository containing the workshop materials is available https://github.com/izahn/R_retrieving_data.

This is an intermediate/advanced R course appropriate for those with basic knowledge of R. If you need a refresher we recommend the Software Carpentry Introductory R material.

Web data aquisition general advice

There are three primary ways that organizations make data available.

1. Bulk download. This is often the best and most convenient option for data analysts.
2. Application Programming Interface (API). This approach typically allows you to search and filter to retrieve specific information. This can be the best option in some cases, but it is also common to resort to using an API because bulk download is not available.
3. Web browser access. This is usually the least desirable, because the interface has been designed to be easy for human beings to navigate, rather than to be easy to parse pragmatically. Usually data analysis will use web scraping techniques as a last resort when no bulk download or API is available.

Example project overview

We are going to retrieve information from https://www.federalregister.gov using both their application programming interface (API) and using web scraping techniques. (We skip bulk downloads because there is nothing to it really. If a bulk download button is available just click it and be done.)

https://www.federalregister.gov is a good example of an organization that provides excellent data access. They provide bulk, API, and browser based access to their data. If we just wanted their data our best bet would be to go to https://www.gpo.gov/fdsys/bulkdata/FR and download it. If we did that we wouldn't learn how to use the API or web interface, so we'll pretend the bulk download doesn't exist and use the API.

The R http client: `httr`

Let's get started by attaching the `httr` package which provides a basic http client in R.

```
library(httr) # for sending GET and POST requests, useful for API's
library(purrr) # for iteration a list manipulation
```

In this section we will examine the basic concepts and tools needed to interact with APIs.

Working with APIs

API Documentation

APIs typically provide documentation describing the available endpoints, the parameters (if any) that each endpoint accepts, and the format of the values the endpoint returns. The federalregister.gov API is documented at https://www.federalregister.gov/developers/api/v1.

Unlike most APIs the federalregister.gov API does not require you to register or obtain an API key. This makes it convenient for use in an API tutorial like this one. One drawback is that all parts of the API use GET requests, so we won't have realistic POST examples.

Get to know your API

There are five basic things you need to know in order to use an API.

1. The **URI** of the resource, e.g., the *URL* providing the service.
2. The **protocol** used to communicate with the resource, e.g., *GET* or *POST*.
3. The **parameters** the resource accepts, e.g., *q* for query, *limit* to specify the number of things to return.
4. The parameter **encoding** the resource expects (e.g., *json*, *xml*, *url-encoded*).
5. The format and structure of the response.

In addition, you *may* need to know what information needs to be sent in the header of your requests, e.g., *authentication* information or *user agent* information. You may also need to pay attention to the *limits* or *restrictions* imposed by the API.

Preparing our first API call

Let's start with the `/documents/facets` endpoint. The base URL is https://www.federalregister.gov/api/v1, so the full path to the resource is https://www.federalregister.gov/api/v1/documents/facets.

The documentation says that it responds to *GET* requests, so we will use the `GET` function provided by the `httr` package. If the documentation indicated that the endpoint responds to *POST* requests we would have used the `POST` function instead.

The documentation indicates that we ask for *daily*, *weekly*, or *monthly* facets (among others). This first parameter just gets tacked on to the end of the URL, so we can access publication counts faceted by day at https://www.federalregister.gov/api/v1/documents/facets/daily, or faceted by month at https://www.federalregister.gov/api/v1/documents/facets/monthly etc.

Make a request to the API using default values

Let's try asking for monthly facets, leaving the remaining parameters at their default values.

```
fr.url <- "https://www.federalregister.gov"
fr.doc.api <- "api/v1"

fr.monthly <- GET(fr.url,
                  path = c(fr.doc.api,
                           "documents",
                           "facets",
                           "monthly"))
```

Examining the request without sending it

Note that the `path` argument is passed to the `modify_url` function – see it's documentation for other arguments that can be passed through the `GET` function. Since GET passes parameters via the URL we can use the `modify_url` function to check that we are constructing the appropriate URL before submitting the request with `GET`.

```
modify_url(fr.url,
                path = c(fr.doc.api,
                         "documents",
                         "facets",
                         "monthly"))
## looks good, so

fr.monthly <- GET(fr.url,
                path = c(fr.doc.api,
                         "documents",
                         "facets",
                         "monthly"))
```

Processing the response

Once we have sent our request and received the response we will usually want to check that it returned successfully, and then parse and process the response.

We can check the status of the response using the `http_status` function.

```
http_status(fr.monthly)
```

If the status is other than "Success", check the URL you are constructing against the documentation and correct it.

Examining the response header

Once we have determined that our request was successful we can proceed to process the response. In order to do that we need to know what format the response is in. Usually this will be documented; the federal register documentation indicates that the response will be in Javascript Object Notation (JSON). If the documentation does not indicate what format the endpoint returns we can inspect the *header* of the response to see if the format is indicated there.

```
headers(fr.monthly)
```

indicates that the response is in *application/json* format.

Extracting the content of the response

The `jsonlite` R package can be used to convert JSON to a native R data structure.

```
library(jsonlite)
```

The specific structure of the *facets* endpoint is not documented, so we'll just have to take a look.

```
## extract and examine content from the response
fr.monthly.content <- content(fr.monthly, as = "text")
cat(str_sub(fr.monthly.content, 1, 100), "\n")

## convert the content to native R list
fr.monthly.content <- fromJSON(fr.monthly.content)
## inspect the result
str(head(fr.monthly.content))
```

If the `as` argument to the `content` function is omitted it will use the `headers` information to guess the content type and parse it for you. If we trust this mechanism we can simply do

```
fr.monthly.content <- content(fr.monthly)
```

to get the content of the response as an R list.

Formating the extracted content

Now we format the data into a convenient form and plot it.

```
fr.monthly.content <- as.data.frame(
  do.call(rbind, map(fr.monthly.content, unlist)),
  stringsAsFactors = FALSE)
str(fr.monthly.content)

library(lubridate)
library(stringr)

fr.monthly.content$name <- dmy(str_c("01",
                                     fr.monthly.content$name,
                                     sep = " "))
fr.monthly.content$count <- as.integer(
  as.character(fr.monthly.content$count))
str(fr.monthly.content)
```

Visualizing the extracted content

```
library(ggplot2)
ggplot(fr.monthly.content, aes(x = name, y = count)) +
    geom_line() +
    scale_x_date(date_breaks = "1 year") +
    theme(axis.text.x = element_text(angle = 60, hjust = 1))
```

Passing parameters

In the previous example we simply asked for the content at https://www.federalregister.gov/api/v1/documents/facets/monthly, leaving the optional parameters at their default values. (These optional parameters are documented at https://www.federalregister.gov/developers/api/v1.) How to we send a request that sets values for these parameters? We format the *query string*, additional information passed in the URL after the *path*. The *query string* starts with *?* and is simply a method of passing additional information to the web service.

Parameter passing example

Let's continue using the `documents/facets` endpoint, but this time we'll facet on *agency*, request counts only for 2015 and only for notices.

```
## first check the url our settings will construct
modify_url(fr.url,
           path = c(fr.doc.api,
                    "documents",
                    "facets",
                    "agency"),
           query = "conditions[publication_date][year]=2015&conditions[type][]=NOTICE")

## looks good, so
fr.agency.2015 <- GET(fr.url,
                 path = c(fr.doc.api,
                          "documents",
                          "facets",
                          "agency"),
                 query = "conditions[publication_date][year]=2015&conditions[type][]=NOTICE")
```

Formating the extracted content

```
## combine the results into a data.frame
fr.agency.2015.content <- as.data.frame(
  do.call(rbind, map(content(fr.agency.2015), unlist))
  )

## cleanup and format
fr.agency.2015.content$count <- as.integer(
  as.character(fr.agency.2015.content$count))

fr.agency.2015.content$name <- reorder(fr.agency.2015.content$name,
                                       fr.agency.2015.content$count)

fr.agency.2015.content <-
  fr.agency.2015.content[order(fr.agency.2015.content$count,
                               decreasing = TRUE), ]
```

Visualizing the extracted content

Now we can visualize the results.

```
ggplot(head(fr.agency.2015.content, 20), aes(x = name, y = count)) +
    geom_bar(stat = "identity") +
    coord_flip()
```

API Exercise

Using the *documents* endpoint at https://www.federalregister.gov/api/v1/documents.json.

1. Use the `GET` function to search for the `conditions[term]` "education". Ask the service to order the results by "newest", limit the result to documents published in 2016, and return 10 documents per page.
2. Check that the request was successful.
3. Extract the content and examine the names.
4. The first element of the result should be "count". Examine the contents of this element to determine how many results matched your request. How would you go about retrieving all the matches?
5. (BONUS) Process the `results` element of your extracted content and extract the `pdf_url`'s. You should end up the a character vector of length 10 containing links to .pdf documents.
6. (BONUS) download each of the 10 pdfs located in step 5.

Use the documents API documentation at https://www.federalregister.gov/developers/api/v1 as needed to determine the required parameters.

If no bulk download or API is available (or the available API is too restrictive) you may have to resort to web scraping. Web scraping is similar to working with an API, except that there is no documentation, and the response will almost always be an html document.

Web Scraping

General web scraping considerations

One important thing to keep in mind is that the http clients in R do not include a javascript engine. This means that you are fine as long as the required logic is executed server-side. If required logic is expected to be evaluated client-side by evaluating javascript your life will be difficult. Your options in that case are a) give up, b) use a local javascript interpreter like PhantomJS, c) write R code to do whatever the javascript is supposed to do, or d) use R to control a web browser (using e.g., the `RSelenium` package) instead of interacting directly with the web service from R. None of the examples we'll look at today require client-side javascript execution, but keep in mind that real-world web scraping may be significantly more complicated because of this.

R packages for web scraping

We could use the `httr` package for web scraping (and I sometimes do) but the `rvest` package contains some convenience functions to make it easier.

```
library(rvest)
```

Web scraping example

Earlier we said that usually if bulk data is available that is preferable to using an API or web scraping to obtain the data. The Federal Register does provide bulk data downloads, with separate downloads for each year at https://www.gpo.gov/fdsys/bulkdata/FR. Even in that case it may be useful to use some web scraping techniques to access all the data. At least it gives us a reasonably simple page that we can use to practice web scraping.

Reading the first page

The first step is to read and parse the starting page at https://www.gpo.gov/fdsys/bulkdata/FR.

```
data.page <- read_html("https://www.gpo.gov/fdsys/bulkdata/FR")
```

Processing the first page

The general idea is that we want to extract links from the first page to figure out where we want to go next. In order to do that we need to be able to identify html elements

CSS selectors

Next we need to identify the elements of the page with the links to the data we want to access. We can use a *css selector* or *xpath* expression to identify element(s) of interest in a web page. In the Chrome browser you can `right-click --> inspect`, then locate the element of interest in the inspector and `right-click --> copy --> copy selector`.

Inspecting the `2000` link tells me that the css selector identifying it is `#bulkdata > table > tbody > tr:nth-child(20) > td:nth-child(1)>a`. Similarly, inspecting the `2017` link tells me that the css selector is `#bulkdata > table > tbody > tr:nth-child(3) > td:nth-child(1) >a`. So we want anchor (`a`) nodes in table rows 3 through 20. I'm going to start by extracting the table, and then extracting all the links from it. We can filter out the ones we don't want later.

Extracting nodes matching a CSS selector

Now that we know how to identify the element we want, we can extract it using the `html_nodes`

```
links.table <- html_nodes(data.page, css = "#bulkdata > table")
links <- html_nodes(links.table, css = "a")
```

Extracting node attributes

Now that we've extracted the anchor elements we can process them and extract the `href` attributes.

```
links <- map_chr(links, html_attr, name = "href")
```

We now have the links we want, but we also have some that we don't want (the first and last). We can filter these out using regular expressions.

```
library(stringr)
links <- str_subset(links, "[0-9]$")
links
```

Prepending the base URL

Finally, we need to append these extracted links to the url they are relative to.

```
links <- str_c("https://www.gpo.gov/fdsys", links, sep = "/")
```

Reading and processing the .html at each link

Now that we have extracted a list of links we want to iterate over them and read the .html they link to. The linked .html pages have other links that we want to locate and extract

Inspecting the next layer of pages

Inspecting https://www.gpo.gov/fdsys/bulkdata/FR/2016 in a web browser suggests that the css selector `#bulkdata > table > tr:nth-child(3) > td:nth-child(1) > a` should give us the link we need. Now we just need to iterate over each link, extract that element, and extract the `href` attribute from it.

Locating and extracting the download links

Let's write a little function that reads the next layer of pages and extracts the data download link.

```
getDataLink <- function(url) {
    page <- read_html(url)
    anchor <- html_nodes(page, css = "#bulkdata > table > tr:nth-child(3) > td:nth-chi
    html_attr(anchor, "href")
}
```

Now we can test our function on the first link.

```
getDataLink(links[[1]])
```

Extracting all the download links

Now that we know how to extract the links, let's get them for all the years.

```
allDataLinks <- str_c("https://www.gpo.gov/fdsys",
                      map_chr(links, getDataLink),
                      sep = "/")
```

The final step of course is to download the data. That part is easy, but we'll skip it in the interest of time.

Web scraping exercise

Start at https://www.gpo.gov/fdsys/bulkdata/FR/2016/12 and use the techniques demonstrated above to extract each to the 12 links to the monthly data for 2016.

Note that css selectors copied from a browser may be incorrect. The selector for the table containing the links should **not** have a `tbody` element.