Wrangle the outcome of an API service

Silvie Cinková

2025-08-12

Table of contents

1 Libraries

```r
library(httr)
library(jsonlite)
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```r
library(readr)
library(magrittr)
library(glue)
library(tidyr)
```

```
Attaching package: 'tidyr'

The following object is masked from 'package:magrittr':

    extract
```

```r
library(purrr)
```

```
Attaching package: 'purrr'


The following object is masked from 'package:magrittr':

    set_names


The following object is masked from 'package:jsonlite':

    flatten
```

2 Data

```r
# url <-
 ↪  "https://api.github.com/repos/open-numbers/ddf--gapminder--systema_globalis/contents/countri
# # Make the GET request
# response <- GET(url)
# write_lines(response,
 ↪  "datasets_ATRIUM//ddf--gapminder--systema_globalis_files_from_GitHubAPI.json")
#
 ↪  read_lines("datasets_ATRIUM//ddf--gapminder--systema_globalis_files_from_GitHubAPI.json",
 ↪  n_max = 2)
#
# filenames_df <- fromJSON(content(response, "text"), flatten = TRUE)
#
# write_tsv(x = filenames_df, file =
 ↪  "datasets_ATRIUM/GitHubURLs_Gapminder_SystemaGlobalis.tsv")
```

3 Explore a GitHub repo of Gapminder

- https://github.com/open-numbers/ddf--gapminder--systema_globalis/tree/master/countries-etc-datapoints

  When you randomly open a few files, it looks as though each file were a table with three columns, the first and second being `geo` (abbreviated country name) and `time` (year), and the third being the social indicator in focus.

- If this is the case, **we can create a one huge table with all indicators by joining the tables**.

Open the URL for illustration. Before, we picked files manually, now we can adopt a more systematic approach.

4 Break the task down

- Get the list of all files from GitHub
- Read in all files
- Check the column names
- make full joins of all tables that have `geo` and `time`
    - Why full joins? We don't know they overlap.
    - Clear of `NA`s later.

We can't tell which countries and which years there are in each file. Depending on the research question, we can operatively filter just non-`NA` rows, or decide that we want to have only countries with a full sequence of years... etc. The immediate goal at this point is to make sure we could make any join at all.

5 Download the list of files from GitHub

- GitHub API: https://api.github.com/

They give you a URL template. You fill in the files location.

https://api.github.com/repos/open-numbers/ddf--gapminder--systema_globalis/contents/countries-etc-datapoints

We will use GitHub's API to automatically retrieve all file names in that repository. Fortunately, GitHub has one that makes this possible. Without it, you would have to try and extract the file names from the html code of the website (*scrape* the website), which would be a much harder task.

> Note
>
> **API (Automated Programming Interface)**: an optional service provided by a website to allow users to interact with the website programmatically.

You want the file names in a folder of a repository, so just give the API the URL of that folder.

6 Call the API

```r
library(httr)
url <- glue("https://api.github.com/repos/open-numbers",
            "/ddf--gapminder--systema_globalis/",
            "contents/countries-etc-datapoints")

# Make the GET request
response <- httr::GET(url)
```

The API sends back a machine-readable version of what you see in the web GUI. Mind that it is not the raw html with the typesetting you would see if you viewed the source code of a website, but a structure that holds just the content of the website as the designers of this API defined it: the metadata of the files in this folder. Usually APIs give you a JSON file.

7 JSON

- **J**ava**S**cript **O**bject **N**otation

```
{"array": [ "object1":}
              {"name1": "property-string/bool",
                "name2": number},
            "object2": {"name222": "property..."}
]
```

You get back a JSON-formatted string that you can save to a file or directly process in R. JSON is one of the standard interchange formats, like e.g. XML or HTML. It originates from JavaScript, and it corresponds to its native data structures, but it became popular across platforms because it is human-readable and technically practical at the same time, and each programming language translates it in its own native structures. For instance, the default interpretation of a JSON file in R is **list**.

Syntax in a nutshell: `[` contains an **array** of **objects** or other arrays. Each object is enclosed in `{` and contains *name-property* (aka *attribute-value*) pairs. Sometimes the names/attributes are also called *keys*. Letter case and punctuation matter, indentation does not.

8 API response

```r
is.list(response)
```

```
[1] TRUE
```

```r
names(response)
```

```
 [1] "url"         "status_code" "headers"     "all_headers" "cookies"    
 [6] "content"     "date"        "times"       "request"     "handle"     
```

This is the response. It has its own class called *response*, but in principle it is just a list. When you save it, To be able to process the list of file names with R, you must first get rid of the header with technical metadata and extract the content (one of the named elements). You would be able to extract it as any other list elements with what you have learned about lists, but `httr` comes with a convenience function called `content`.

9 Extract the content part of an API response

```r
httr::content(response) %>% class()
```

```
[1] "list"
```

```r
httr::content(response) %>% length()
```

```
[1] 490
```

```r
httr::content(response) %>% names()
```

```
NULL
```

This would be the default translation of a JSON file: the array translates to a list. If you look at the original, you will find out that each file name along with its metadata is one JSON object in an array, but these objects are not named. Let us take a closer look at the first element and hope that all have the same structure.

10 Structure of a file name item

```
content(response)[[1]] %>% class()
```

```
[1] "list"
```

```
content(response)[[1]] %>% names()
```

```
 [1] "name"         "path"         "sha"          "size"         "url"         
 [6] "html_url"     "git_url"      "download_url" "type"         "_links"      
```

```
content(response)[[1]][1]
```

```
$name
[1] "ddf--datapoints--adults_with_hiv_percent_age_15_49--by--geo--time.csv"
```

```
content(response)[[1]]$`_links` %>% class()
```

```
[1] "list"
```

You subset a list to extract an element either by its position index in double square brackets (`[[1]]`) or, when available, by its name. When that name is weird (e.g. starts with an underscore), enclose it in back ticks like when accessing a weird column name in a data frame. That JSON item looked a bit more complex in the original, and hence it comes as no surprise that it translated as a nested list inside an element of the content list. It is not just a name-property pair, but a named object with three name-property pairs.

> Tip
>
> Recap what else you know about lists: You can access only one element at a time. When you want to make a larger selection, you must create a subset of the list, that is, to make a smaller list with the selected elements. And then access the individual elements one by one again.

11 Parse JSON vs. Wrangle the list

- two valid approaches
- depends on JSON complexity
- maybe we can drop something

These are two valid approaches. Perhaps the more convenient one is parsing the JSON and that is what we will do, but it is still useful to know that if there is an issue with JSON parsing, we can fall back to working with an ordinary list!

12 Library `jsonlite`

- Read a character vector: `from_JSON`
  - we have a list but `httr::content` can make it a vector
  - all source JSON in one element

```
response_vec <- content(response, type = "text")
```

```
No encoding supplied: defaulting to UTF-8.
```

```
class(response_vec); length(response_vec)
```

```
[1] "character"
```

```
[1] 1
```

13 Parse JSON from a character vector

```
filenames_df <- fromJSON(response_vec, flatten = TRUE)
colnames(filenames_df)
```

```
 [1] "name"         "path"         "sha"          "size"         "url"
 [6] "html_url"     "git_url"      "download_url" "type"         "_links.self"
[11] "_links.git"   "_links.html"
```

```
write_tsv(filenames_df, "datasets_ATRIUM/gapminder_metadata_filenames.tsv")
```

By default, `fromJSON` simplifies the structures to fit them to a data frame. You can override it, when you need to just change something in the JSON and return the same structure back, but for our purposes it is the ideal. We have a data frame!!!

If you want to save it in a file, you must also flatten it. Before, it contained a column with vectors inside. This is possible for tibbles, but not csv files. We will drop this column anyway.

14 Simplify the structure

```
filenames_df %<>% select(c(name, download_url, size))
filenames_df %<>% slice(1:10)
# we take just 10 rows for a proof of concept
```

We only need the name and download url, perhaps size.

What do we ask again?

- file has columns `geo` and `time` : logical columns
- how many columns : numeric column
- maybe retrieve the column names of each file for future reference

At any rate we will have to read the files, at least one row to get the column names. One very transparent way is to set up a loop and successively fill four objects: one vector for the geo column, one vector for the time column, one vector for the number of columns, and a list of vectors with column names. Then we can add them as columns (yes, a column name of a data frame can contain a vector, see above). Keywords to this topic: `tidyr`, `enframe`, `unnest_longer`, `unnest_wider`.

15 Check it out for one file

```
my_filecolnames <- read_csv(file =
                               filenames_df$download_url[1],
                             n_max = 1) %>%
  colnames()
```

```
Rows: 1 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): geo
dbl (2): time, adults_with_hiv_percent_age_15_49

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
geo_column <- "geo" %in% my_filecolnames
time_column <- "time" %in% my_filecolnames
colnames_length <- length(my_filecolnames)
```

16 Setting up the loop 1

- prepare empty objects before!
- sequence: filenames

```
filenames <- filenames_df$name
urls <- filenames_df$download_url
has_geo <- logical()
has_time <- logical()
colnames_length <- numeric()
file_colnames <- list()
# Only now we can set up the loop
```

The idea with the loop is that each loop delivers four objects, which we collect in exactly the same order as we input the filenames into the loop. First we need to set up these objects as sort of empty shells and incrementally fill them with the outputs of the loop. The empty objects must exist before the loop, and the adding is done inside the loop. The most efficient way is to subset the object with the index of the loop iterator (that mysterious *i*) and assign it a value that we get from the script inside the loop.

17 The loop syntax explained

```
for (i in seq_along(filenames)) {

}
```

You have a sequence on which you want to run a piece of code. That sequence is the vector of file names. `seq_along` produces a vector with a numeric sequence starting with one and ending with the position index of the last element of `filenames`. So it ought to have 10 elements, because we sliced 10 rows from `filenames_df`.

```
seq_along(filenames)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Now, `i` becomes `1` in the first run, `2` in the second, and so on, until it finishes when it has run through the code as `10`. It is going to work for us as a subsetting index.

18 The loop filled

```
for (i in seq_along(filenames)) {
  my_filecolnames <- read_csv(file =
                               filenames_df$download_url[i],
                             n_max = 1, show_col_types = FALSE) %>%
  colnames()
geo_column <- "geo" %in% my_filecolnames
time_column <- "time" %in% my_filecolnames
# now add these results as new elements to the objects!
has_geo[i] <- geo_column
has_time[i] <- time_column
colnames_length[i] <- length(my_filecolnames)
file_colnames[[i]] <- my_filecolnames
}
```

Note the `[i]` in `read_csv`. The variable `my_filecolnames` is going to result from a different file in each run!

19 Add them to the data frame

```
metadata_explored <- filenames_df %>%
  bind_cols(has_geo = has_geo,
            has_time = has_time,
            colnames_length = colnames_length) %>%
  mutate( file_colnames = file_colnames) %>%
  tidyr::unnest_wider(file_colnames, names_sep = "_")
```

20 Without loop, use `purrr::map`

```
get_details <- function(url) {
  my_filecolnames <- readr::read_csv(file = url, n_max = 1, show_col_types =
 ↪  FALSE) %>%
    colnames()
  geo_column <- "geo" %in% my_filecolnames
  time_column <- "time" %in% my_filecolnames
  colnames_length <- length(my_filecolnames)
  tibble(
    "url" = url,
    "has_geo" = geo_column,
    "has_time" = time_column,
    "colnames_length" = colnames_length,
   "file_colnames" = my_filecolnames )
}
```

```
otest <- purrr::map(filenames_df$download_url, ~ get_details(.x)) %>%
 ↪  list_rbind()
#otest %<>% filter(!(file_colnames %in% c("geo", "time")))
metadata_explored2 <- left_join(filenames_df, otest, by = c("download_url" =
 ↪  "url"))
```

21 Metadata explored

```
metadata_explored %>% slice(1:2) #%>% kableExtra::kable()
```

```
# A tibble: 2 x 9
  name        download_url  size has_geo has_time colnames_length file_colnames_1
  <chr>      <chr>        <int> <lgl>   <lgl>              <dbl> <chr>
1 ddf--data~ https://raw~ 43902 TRUE    TRUE                   3 geo
2 ddf--data~ https://raw~  9839 TRUE    TRUE                   3 geo
# i 2 more variables: file_colnames_2 <chr>, file_colnames_3 <chr>
```

```
metadata_explored2 %>% slice(1:2) #%>% kableExtra::kable()
```

```
                                                                   name
1 ddf--datapoints--adults_with_hiv_percent_age_15_49--by--geo--time.csv
2 ddf--datapoints--adults_with_hiv_percent_age_15_49--by--geo--time.csv

1 https://raw.githubusercontent.com/open-numbers/ddf--gapminder--systema_globalis/master/cou
2 https://raw.githubusercontent.com/open-numbers/ddf--gapminder--systema_globalis/master/cou
   size has_geo has_time colnames_length file_colnames
1 43902    TRUE     TRUE               3           geo
2 43902    TRUE     TRUE               3          time
```

22 Try out mutate

- still with my function but create a list
- `rowwise` is the clue
- use `list` before the function even if it returns a list

```
get_details_list <- function(url) {
  my_filecolnames <- readr::read_csv(file = url, n_max = 1, show_col_types =
↪  FALSE) %>%
    colnames()
  geo_column <- "geo" %in% my_filecolnames
  time_column <- "time" %in% my_filecolnames
  colnames_length <- length(my_filecolnames)
 list(
    "url" = url,
    "has_geo" = geo_column,
    "has_time" = time_column,
    "colnames_length" = colnames_length,
   "file_colnames" = my_filecolnames )
}
```

```
filenames_df %>%
  rowwise() %>%
  mutate(newcol = list(get_details_list(download_url))) %>%
  unnest_wider(newcol, names_sep = "_") %>%
  unnest_wider(newcol_file_colnames, names_sep = "__") %>%
  ungroup() #%>% kableExtra::kable()
```

```
# A tibble: 10 x 10
   name             download_url  size newcol_url newcol_has_geo newcol_has_time
   <chr>            <chr>        <int> <chr>      <lgl>          <lgl>
 1 ddf--datapoints~ https://raw~ 43902 https://r~ TRUE           TRUE
 2 ddf--datapoints~ https://raw~  9839 https://r~ TRUE           TRUE
 3 ddf--datapoints~ https://raw~ 81653 https://r~ TRUE           TRUE
 4 ddf--datapoints~ https://raw~ 42723 https://r~ TRUE           TRUE
 5 ddf--datapoints~ https://raw~ 84589 https://r~ TRUE           TRUE
 6 ddf--datapoints~ https://raw~ 84550 https://r~ TRUE           TRUE
 7 ddf--datapoints~ https://raw~ 87350 https://r~ TRUE           TRUE
 8 ddf--datapoints~ https://raw~ 65710 https://r~ TRUE           TRUE
 9 ddf--datapoints~ https://raw~ 84546 https://r~ TRUE           TRUE
10 ddf--datapoints~ https://raw~ 40953 https://r~ TRUE           TRUE
# i 4 more variables: newcol_colnames_length <int>,
#   newcol_file_colnames__1 <chr>, newcol_file_colnames__2 <chr>,
#   newcol_file_colnames__3 <chr>
```

23 We cannot extract the info from `str` nor `summary`

```
summary_info <- summary(list(a = 1, b = 2, c = 3))
summary_info
```

```
  Length Class  Mode
a 1      -none- numeric
b 1      -none- numeric
c 1      -none- numeric
```

```
str(summary_info)
```

```
 'summaryDefault' chr [1:3, 1:3] "1" "1" "1" "-none-" "-none-" "-none-" ...
 - attr(*, "dimnames")=List of 2
```

```
  ..$ : chr [1:3] "a" "b" "c"
  ..$ : chr [1:3] "Length" "Class" "Mode"
```

```
class(summary_info)
```

```
[1] "summaryDefault" "table"
```