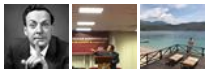
R news and tutorials contributed by hundreds of R bloggers

Welcome!

Here you will find daily **news and tutorials about R**, contributed by hundreds of bloggers. There are many ways to **follow us -**

By e-mail:

On Facebook:



If you are an R blogger yourself you are invited to add your own R content feed to this site (**Non-English** R bloggers should add themselves- here)

Jobs for R-users

- Senior Research Specialist II
- Fisheries Analyst/Senior Fisheries Analyst
- Senior Scientist, Translational Informatics @ Vancouver, BC, Canada
- Senior Principal Data Scientist @ Mountain View,

Recent Posts

abstracts
- My newyeaRs resolution: slimming down (Seurat)
- rOpenSci 2019 Code of Conduct Transparency Report

Other sites

- SAS blogs
- Jobs for R-users

Accessing APIs from R (and a little R programming)

November 26, 2015
By Christoph Waldhauser

Working with APIs

Christoph Waldhauser

22/11/2015

APIs are the driving force behind data mash-ups. It is APIs that allow machines to access data programmatically – that is automatically from within a program – to make use of API provided functionalities and data. Without APIs much of today's Web 2.0, Apps and data applications would be outright impossible.

This post is about using APIs with R. As an example. we'll use the EU's EurLex[1] data base API as provided by Buhl Rassmussen. This API is a good example of the APIs you might find in the wild. Of course, there are the APIs of large vendors, like Google or Facebook, that are thought out and well documented. But then there is the vast majority of smaller APIs for special applications that often lack in structure or documentation. Nevertheless, these APIs often provide access to valuable ressources.

Background on APIs

API is short for Application Programming Interface. Basically, it means a way of accessing the functionality of a program from inside another program. So instead of performing an action using an interface that was made for humans, a point and click GUI for instance, an API allows a program to perform that action automatically. The power of this concept becomes only visible, when you imagine that you can mesh the calling of an API in the program with anything else that program might want to do. Some examples from data science:

- Retrieve data and produce a visualization from it that gets updated every time someone looks at it
- Have tweets automatically translated and entities reported
- Have additional nodes in a computer cluster launched as soon as tasks become cumbersome, to ensure fast data processing

While an API can be any defined interface between two programs, today APIs usually refer to a special kind of APIs that are based on the WWW's HyperText Transfer Protocol (HTTP) that is also used by web servers and browsers to exchange data. Indeed, one might consider

browsing the web as using APIs: a program (the browser) uses a defined set of commands and conventions to retrieve data (the webpage) from a remote server (the website) and renders it locally in the browser (the thing you see).

All web-based [2] APIs have always the same structure: they consist of a `URL` to a domain and a `path` to an endpoint. For instance: `http://example.com/api` where `http://example.com` is the `URL` and `/api` is the path to the endpoint.

Web-based APIs that are used for data science come usually in two flavors that are named after the HTTP verbs defined [3]:

- `GET` – sends a set of parameters, a `query` to an endpoint and then recieves an answer.
- `POST` – sends a data payload to an endpoint to be processed at the remote system, usually receiving only a success message as an answer.

There are other verbs defined in `HTTP`, like `DELETE`, but they are less common in APIs. The by far largest group of APIs makes only use of the `GET` verb. Let's look at that flavor in greater detail.

A canonical example would be an API that allows to retrieve data from some data base and the API's `query` can be used to narrow down the selection. Let's say an API provides access to newspaper articles. By specifying the parameter `year` the API returns not all articles, but only those that were written in a specific year. Let's say we are only interested in articles written in 2014. The corresponding API call would, thus, look like: `http://example.com/api?year=2014`. We already know which part is the `URL` and which part is the `path` to the endpoint. What's new is the query `year=2014`. Note that it's separated from the `path` by a question mark. In this example, `year` is the name of the parameter, and `2014` is its value.

Upon receiving the API call, the remote system crafts an answer. The answer can be in any format. It could be a image file, or a movie, or text, or … In recent years, JSON has become the most common answer format by far. JSON is a simple text file that uses special characters and conventions to bring structure into its contents. You can

find more info at the Wikipedia page on JSON. For now it suffices to know that is a popular format to store data, that can potentially be nested and delivered together with metadata. And that R can process it quite easily.

The big problem with APIs is that they are always designed by humans. So APIs vary wildly in logical structure and the quality of documentation. This unfortunately means, that there is no simple catch-all solution for working with APIs and all programs will need to be custom tailored to the API used. This also means that using an API almost always requires programming to some degree.

Accessing APIs from R

In this example, we'll use R to retrieve data from an API and process it. The API we'll query provides data on EU legislative documents. More specifically, we are interested in which week day is most popular for EU energy legislative documents to go into force. Ok, perhaps that's not a mind blowing research question, but one that will allow us to demonstrate the using of APIs and the required data processing quite nicely.

Required packages

There are many facilities in R that can be used to access APIs. The one package that I find most useful is Hadley[4]'s `httr`. It allows for easy crafting of API calls and also handling the more intricate aspects of APIs like authentication.

Working with JSON data is facilitated a lot by the `jsonlite` package. It does a good job translating JSON's nested data structures into sensible R objects. Well, most of the time, anyway.

Since in this example we are going to work with dates, let's use another of Hadley's packages: `lubridate`. If you work with dates frequently, it's a package that might be a valuable addition to your toolbox.

If you don't yet have these packages installed, you can use this R code to obtain them:

```
install.packages(c("httr", "jsonlite", "lubridate"))
```

Outline of the example

In this subsection I'll outline the steps required to perform to find an answer to our question (which is the most popular day for having energy related documents turn into force?).

EurLex documents all bear a document classifier (directory code in EurLex parlance) that can be used to single out documents that relate to a specific topic. The EurLex classifiers are always four dot separated pairs of digits. For instance, the classifier `07.40.30.00` identifies documents that relate to air traffic safety. We will use the appropriate classifiers to retrieve the data on energy related documents.

So, these are the required steps we'll need to take to get our answer:

1. Retrieve a list of all the valid classifiers
2. Extract from that answer only those classifiers that relate to energy, i.e. start with `12..`[5]
3. Retrieve the documents' meta data that are classified with one of the classifiers we've found to be relevant.
4. Work with the data we've retrieved to find out which weekday is most frequent.

Steps (1) and (3) will involve calling the API and (2) and (4) are just local data processing chores.

Before doing anything else, we need to load the required packages:

```
library(httr)
library(jsonlite)
```

```
##
## Attaching package: 'jsonlite'
##
## The following object is masked from 'package:utils':
```

```
##
##     View

library(lubridate)
```

One more thing before we get started: R has the “feature” of turning character strings automatically into factor variables. This is great, when doing actual statistical work. It is this magic that allows R to turn multinomial variables into dummy variables in regression models and produce nice cross tables. When working with APIs, however, this “feature” becomes a hinderance. Let’s just turn it off. Note: this call only affects the current session; when you restart R, all settings will be back to normal.

```
options(stringsAsFactors = FALSE)
```

Retrieving valid classifiers

Calling the `/eurlex/directory_code` endpoint directly, retrieves a list of all valid classifiers. Let’s obtain that list. First, we need set up the `URL` and `path` part of the API call. A query is not required at this point, as the API provides the answer directly.

```
url  <- "http://api.epdb.eu"
path <- "eurlex/directory_code"
```

Executing an API call with the `GET` flavor is done using the `GET()` function.

```
raw.result <- GET(url = url, path = path)
```

Let’s explore what we’ve got back:

```
names(raw.result)
```

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

The result we got back from the API is a list of length 10. Of these, two parts are important:

- `status_code` that tells us, if the call worked network-wise. For a list of possible status codes, see https://en.wikipedia.org/wiki/List_of_HTTP_status_codes.
- `content` the API's answer in raw binary code, not text. Alas, the answer could also be an image or a sound file.

If we examine the status code,

```
raw.result$status_code
```

```
## [1] 200
```

we see that we've got `200`, which means, all worked out fine. Note that this status code only tells us, that the server recieved our request, not if it was valid for the API or found any data.

Let's look at the actual answer or data payload we've got back. Let's just look at the first few elements:

```
head(raw.result$content)
```

```
## [1] 7b 22 30 31 2e 30
```

That's useless, unless you speak Unicode. Let's translate that into text.

```
this.raw.content <- rawToChar(raw.result$content)
```

Let's see how large that is in terms of characters:

```
nchar(this.raw.content)
```

```
## [1] 121493
```

That's rather large. Let's look at the first 100 characters:

```
substr(this.raw.content, 1, 100)
```

```
## [1] "{"01.07.00.00":{"directory_code":"01.07.00.00","url":"http:\/\/api.epdb.eu\/eurlex\/directory_code\/"
```

So the result is a single character string that contains a JSON file. Let's tell R to parse it into something R can work with.

```
this.content <- fromJSON(this.raw.content)
```

What did R make out of it?

```
class(this.content) #it's a list
```

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

```
length(this.content) #it's a large list
```

```
## [1] 462
```

```
this.content[[1]] #the first element
```

```
## $directory_code
## [1] "01.07.00.00"
##
## $url
## [1] "http://api.epdb.eu/eurlex/directory_code/?dc=01.07.00.00&key="
##
## $number_of_documents
## [1] "126"
##
## $list_of_acts_inforce_eurlex
## [1] "http://eur-lex.europa.eu/Result.do?RechType=RECH_repertoire&rep=0107*&repihm="
```

```
this.content[[2]] #the second element
```

```
## $directory_code
## [1] "01.10.00.00"
##
## $url
## [1] "http://api.epdb.eu/eurlex/directory_code/?dc=01.10.00.00&key="
##
## $number_of_documents
## [1] "191"
##
## $list_of_acts_inforce_eurlex
## [1] "http://eur-lex.europa.eu/Result.do?RechType=RECH_repertoire&rep=011*&repihm="
```

So, apparently R makes a list out of it, with one element per classifier.
Each element has:

- the directory code document classifier
- a URL where one can retrieve more details
- the number of documents with that classifier
- another URL with yet more details

So, essentially, the result is a list of lists. Lists are not (always) very nice to work with, and lists of lists are usally despicable. Let's turn it into a data frame:

```
this.content.df <- do.call(what = "rbind",
                           args = lapply(this.content, as.data.frame))
```

This call does a number of things. `lapply(this.content, as.data.frame)` turns each of the 462 list elements into mini single-row data frames. This is required, so that we then can combine (`rbind`) them all together into a single data frame.

In case you are interested in the gory details:

1. The function `rbind` takes any number of data frames as arguments, and turns them into a single data frame, by just stacking one on top of the next. So `C <- rbind(A, B)` will yield a data frame that has first the contents of `A` and then those of `B` stacked on top of each other.
2. The function `lapply` takes a list (the first argument) and applies the function that is the second argument (here: `as.data.frame`) to each of its elements. So, the call to `lapply` turns our list of lists into a list of single row data frames.
3. The function `do.call` is a true wonder girl. She uses its `args` argument as **arguments** to the function named at the `what` argument. So here, it executes `rbind` with all the elements (single row data frames, that we created in (2)) of our list of data frames. It is the same as typing:
   `rbind(OurDfList[[1]], OurDfList[[2]], OurDfList[[3]], ...)`
   where `...` would need to be replaced with all the other list items.

What have we got now?

```
class(this.content.df) #a single data frame

## [1] "data.frame"

dim(this.content.df)   #with 462 rows and 4 variables

## [1] 462   4

head(this.content.df)
```

```
##             directory_code
## 01.07.00.00    01.07.00.00
## 01.10.00.00    01.10.00.00
## 01.20.00.00    01.20.00.00
## 01.30.00.00    01.30.00.00
## 01.40.00.00    01.40.00.00
## 01.40.10.00    01.40.10.00
##                                                                  url
## 01.07.00.00 http://api.epdb.eu/eurlex/directory_code/?dc=01.07.00.00&key=
## 01.10.00.00 http://api.epdb.eu/eurlex/directory_code/?dc=01.10.00.00&key=
## 01.20.00.00 http://api.epdb.eu/eurlex/directory_code/?dc=01.20.00.00&key=
## 01.30.00.00 http://api.epdb.eu/eurlex/directory_code/?dc=01.30.00.00&key=
## 01.40.00.00 http://api.epdb.eu/eurlex/directory_code/?dc=01.40.00.00&key=
## 01.40.10.00 http://api.epdb.eu/eurlex/directory_code/?dc=01.40.10.00&key=
##             number_of_documents
## 01.07.00.00                 126
## 01.10.00.00                 191
## 01.20.00.00                  69
## 01.30.00.00                  24
## 01.40.00.00                 382
## 01.40.10.00                 514
##                                                    list_of_acts_inforce_eurlex
## 01.07.00.00  http://eur-lex.europa.eu/Result.do?RechType=RECH_repertoire&rep=0107*&repihm=
## 01.10.00.00   http://eur-lex.europa.eu/Result.do?RechType=RECH_repertoire&rep=011*&repihm=
## 01.20.00.00   http://eur-lex.europa.eu/Result.do?RechType=RECH_repertoire&rep=012*&repihm=
## 01.30.00.00   http://eur-lex.europa.eu/Result.do?RechType=RECH_repertoire&rep=013*&repihm=
## 01.40.00.00   http://eur-lex.europa.eu/Result.do?RechType=RECH_repertoire&rep=014*&repihm=
## 01.40.10.00 http://eur-lex.europa.eu/Result.do?RechType=RECH_repertoire&rep=01401*&repihm=
```

That's nice and we can work with it to extract all the classifiers of energy topics.

Extracting energy classifiers

We've almost found the classifiers for energy topics. We just need to filter them out of all the other classifiers that are available. Remember, Energy classifiers start with `12.`. We can use that fact:

```
headClass <- substr(x     = this.content.df[, "directory_code"],
                    start = 1,
                    stop  = 2)
```

`headClass` is now just a character vector containing the first two characters of the `directory_code` for each of the 462 different

classifiers.

```
length(headClass)
```

```
## [1] 462
```

```
head(headClass)
```

```
## [1] "01" "01" "01" "01" "01" "01"
```

If these first two characters equal `12`, it's an energy topic:

```
isEnergy <- headClass == "12"
table(isEnergy) # 19 of the topic classifiers start with 12
```

```
## isEnergy
## FALSE  TRUE
##   443    19
```

Let's use this logical vector to index our data frame:

```
relevant.df <- this.content.df[isEnergy, ]
```

And let's narrow that down to solely the document identifiers:

```
relevant.dc <- relevant.df[, "directory_code"]
```

`relevant.dc` is now a character vector with all the directory codes that are relating to energy topics.

```
length(relevant.dc)
```

```
## [1] 19
```

```
relevant.dc
```

```
##  [1] "12.07.00.00" "12.10.00.00" "12.10.10.00" "12.10.20.00" "12.20.10.00"
##  [6] "12.20.20.00" "12.20.30.00" "12.20.40.00" "12.30.00.00" "12.40.00.00"
## [11] "12.40.10.00" "12.40.20.00" "12.40.30.00" "12.40.40.00" "12.40.50.00"
## [16] "12.50.10.00" "12.50.20.00" "12.50.30.00" "12.60.00.00"
```

We've come a long way. We've retrieved all possible classifiers from the API, parsed the answer so that we can work with it, and, finally, extracted all those classifiers that are relating to energy topics.

Retrieving energy documents' meta data

In this step, we use the identified classifiers to retrieve all the documents' meta data that relate to energy topics. We'll use the API again. The base parts of the API call have not changed.

Now to the query: we cannot pass all the relevant classifiers in a single call. Rather, we need to create 19 queries, one for each identified classifier. There are many ways to do that. Let's do it the pretty way with our own function:

```
makeQuery <- function(classifier) {
  this.query <- list(classifier)
  names(this.query) <- "dc"
  return(this.query)
}
```

Remember, that a the query part of an API call is a named list. The name is the name of the API parameter, and it's value is, well, it's value. Our function, `makeQuery()` takes a single argument, `classifier` and turns it into a single element list and sets the name of that list's single element to be `dc`. Let's try it out with nonsense

```
makeQuery("foo")
```

```
## $dc
## [1] "foo"
```

We discover, that the function indeed returns a list with a single element, that element is named `dc` and it's value is the string we've specified.

Let's apply our new function to all of our relevant classifiers from above to turn them into queries. Remember the `lapply` function we can use to apply a function to each element of a list:

```
queries <- lapply(as.list(relevant.dc), makeQuery)
```

Now we have a list (`queries`) that is composed of all the individual queries that result from our function. Now we are good to go to acutally execute the API calls.

It's time now to execute the API calls we've created. Let's start out with the first element of our query list.

```
this.raw.result <- GET(url = url, path = path, query = queries[[1]])
```

What did we get back?

```
this.result <- fromJSON(rawToChar(this.raw.result$content))
```

We've got back the meta data for all 11 documents that are classified with our first relevant classifier (`12.07.00.00`). For each document, we get:

```
names(this.result[[1]])
```

```
##  [1] "form"                "title"               "api_url"
##  [4] "eurlex_perma_url"    "doc_id"              "date_document"
##  [7] "of_effect"           "end_validity"        "oj_date"
## [10] "directory_codes"     "legal_basis"         "addressee"
## [13] "internal_ref"        "additional_info"     "text_url"
## [16] "prelex_relation"     "relationships"       "eurovoc_descriptors"
## [19] "subject_matter"
```

Apparently, our call does work just fine. Executing the first query stored in `queries` results in 11 documents and their respective meta data.

Let's execute the query for each element in `queries`. How to go about this? Why not use `lapply` to execute each of the queries in the list? We could do that, but let's try another approach: a loop. A loop – loops – over a set of numbers, and at each iteration executes some code. Let's try that:

First, we need something where we can store the results. For that, we create an empty list with just enough room to store each of the queries' results:

```
all.results <- vector(mode   = "list",
                      length = length(relevant.dc))
```

This made a new, empty list called `all.results`. It has as many empty slots as we have energy related classifiers.

```
for (i in 1:length(all.results)) {
  this.query       <- queries[[i]]
  this.raw.answer  <- GET(url = url, path = path, query = this.query)
  this.answer      <- fromJSON(rawToChar(this.raw.answer$content))
  all.results[[i]] <- this.answer
  message(".", appendLF = FALSE)
  Sys.sleep(time = 1)
}
```

This loop iterates over the numbers 1 to 19 (the number of relevant classifiers). At each iteration, it:

- loads the appropriate query (whos time has come)
- executes the API call with that query
- extracts the content of the response and converts it from JSON
- writes the results to the empty list we had created beforehand
- prints a dot, so we don't get bored waiting
- waits for a second (because we are polite and don't want to bog down EU)

`all.results` is now no empty list no more. It is filled with the answers the API has produced as result to our 19 queries.

Now we have results. The next step is to beat these results into a shape we can actually use to find our corvetted answer. Of all the parts of the answer, we are interested in `form`, `date_document` and `of_effect`. Let's create another function that returns just these parts as a data frame.

```
parseAnswer <- function(answer) {
  this.form   <- answer$form
  this.date   <- answer$date
  this.effect <- answer$of_effect
  result <- data.frame(form   = this.form,
                       date   = this.date,
                       effect = this.effect)
  return(result)
}
```

Let's try our function on one of the results. Remember, that the results we've retrieved is a list (19 elements, one for each classifier) of lists (one for each document; the numbers of documents varies from classifier to classifier).

```
parseAnswer(all.results[[1]][[2]])
```

This took from the first classifier (the `[[1]]`) the second document (the `[[2]]`). We see, it's a data frame with one row and three columns.

Now to apply our function to all of our list of lists results, we need to go a little deeper and combine all the skills we've learned so far. We could use a loop to iterate over all the 19 classifiers first and then a second loop to iterate over all the documents in each classifier. But that's rather verbose and cumbersome. Let's use `lapply` instead:

```
parsedAnswers <- lapply(all.results,
                        function(x) do.call("rbind", lapply(x, parseAnswer)))
```

What's happening here? Let's start from the inside out:

1. We apply our function, `parseAnswer` on each document in a classifier
2. Inside each classifier, we `rbind` the single line data frames together to form a single data.frame with one row per document.
3. We do this for each of the 19 classifiers in `all.results`.

We get a back a list of data frames, each data frame having as many rows as there are documents in that classifier.

```
class(parsedAnswers) #list
```

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

```
length(parsedAnswers) #19
```

```
## [1] 19
```

```
sapply(parsedAnswers, nrow) #11, 15, 107, ...
```

```
##  [1]  11  15 107 110 172  41  16  22  55  42  16  60  62 143  84  18  11
## [18]  65  28
```

Let's combine these 19 data frames in a single one. How can we do that? Of course just like before using `do.call` and `rbind`.

```
finalResult <- do.call("rbind", parsedAnswers)
class(finalResult) #data.frame
```

```
## [1] "data.frame"

dim(finalResult) # 1078 rows, 3 columns

## [1] 1078    3
```

All the final results are now contained neatly in a single data frame. Note that the data frame's row names are actually the document IDs. We can use them to retrieve the actual document's meta data.

Working with dates

The data we've retrieved from the API is still all only characters. If we tell R that the date columns (`date` and `effect`) are actually dates, R can calculate with these dates.

First, we need to convert characters to dates. Let's try this out with some arbitrary date.

```
date.character <- "1981-05-02"
date.POSIXct <- ymd(date.character)

class(date.character) #character

## [1] "character"

class(date.POSIXct) #POSIXct

## [1] "POSIXct" "POSIXt"
```

That worked just fine. Let's do this for the date columns in our final results data frame:

```
finalResult$date <- ymd(finalResult$date)
finalResult$effect <- ymd(finalResult$effect)
```

At last, we've retrieved all the data we need to answer our question and brought it into a format we can work with. Let's answer our question, which day of the week is most popular for letting laws become effective:

```
finalResult$effectDay <- wday(finalResult$effect, label = TRUE)
table(finalResult$effectDay) #Most documents went into effect on a Wednesday
```

```
##
##   Sun   Mon  Tues   Wed Thurs   Fri   Sat
##    31   160   132   172   145   384    54
```

We see, that Wednesdays are the most popular ones.

This concludes this little tour de force of introducing working with APIs with R. We covered not only how to craft API calls, but also how to use R's (list) programming features to deal with API answers and beat data into shape.

---

1. EurLex documents have been used in the past as text-book examples for statistical programming and machine learning. See for instance TU Darmstadt's project.↩
2. well, most anyway↩
3. Actually, it's a little bit more complicated. `GET` and `POST` are methods an API might implement. Often, the same API will provide `GET` and `POST` methods side by side for different purposes. If taking the HTTP standard as the proverbial letter of the law, `GET` methods should not change anything on the remote system, i.e. only return data, while `POST` methods should change a state or a file on the remote system. In practice, `POST` methods are also used to provide data to the remote system that it can use to work with, e.g. a text that should be automatically translated. ↩
4. Hadley Wickham is perhaps one of the most prolific R developers. He's responsible for a great wealth of packages, among them the visualization package `ggplot2` and the data munging facilities of `dplyr`. Check out Hadley's personal website to get a glimpse on all the projects he's involved with. ↩
5. Finding out that `12.` is the document classifier that identifies energy related documents actually took quite some research. For easy access, Project Mulan provides a list of all EurLex directory codes.↩

Search R-bloggers

Most visited articles of the week

8. Date Formats in R
9. In-depth introduction to machine learning in 15 hours of expert videos

Sponsors

Jobs for R users

- Senior Research Specialist II
- Fisheries Analyst/Senior Fisheries Analyst
- Senior Scientist, Translational Informatics @ Vancouver, BC, Canada
- Senior Principal Data Scientist @ Mountain View, California, United States
- Technical Research Analyst – New York, U.S.
- Movement Building Analyst
- Business Intelligence Analyst