

Week 8: APIs

LSE MY472: Data for Data Scientists

<https://lse-my472.github.io/>

Autumn Term 2024

Ryan Hübert

Introduction

- We've now learned how to **take** data from structured, unstructured, and even dynamic webpages
- However, many sources prefer to simply **give** us (some of) their data(!)
- This week we will learn about the modal way of doing this – web APIs

Plan for today

- APIs: the big picture
- JSON
- API calls

APIs: the big picture

APIs

- API: **A**pplication **P**rogramming **I**nterface
- API user sends a request to the API (e.g. through R) and the API returns data from the API provider's database, in accordance with the provider's permissions
- Key: APIs don't have a “**user interface**” although they can be connected to one
- APIs are widely used to communicate between applications
- In **web APIs**, a set of structured HTTP/S requests can return data in a lightweight format e.g. JSON or XML
 - Simple example: Google Maps Geocode API returns latitude and longitude for a location
- APIs are now also widely available for data-curious scientists

APIs vs. Scraping for Data Science

Advantages

- Cleaner data collection: Avoid malformed HTML, fewer legal issues, clear data structures, more trust in data collection. . .
- Standardised data access procedures: Transparency, replicability
- Robustness: Many users/developers is usually a good thing, support may exist

Disadvantages

- Not always available
- Dependency on API providers (e.g. Twitter/X)
- Rate limits
- Price

APIs for Social Media

One area that APIs have been used extensively for in social science is the study of social media

A lot of work was done with Twitter's REST and Streaming APIs, but these are now essentially defunct

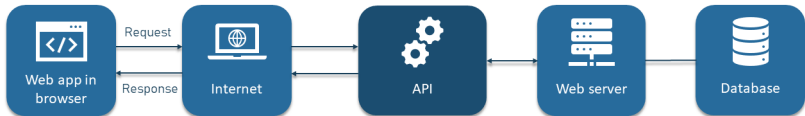
Other social media APIs do exist, but the taps are being shut:

- Facebook/Meta have limited APIs
- Reddit (8-12 weeks for researcher approval)
- Bluesky (very new)
- etc.

It's a big problem for researchers!

How (web) APIs work

HOW API WORKS



Source: <https://www.altexsoft.com/blog/what-is-api-definition-types-specifications-documentation/>

Working with web APIs

Types of APIs:

- **RESTful APIs**: Queries for static information at current moment (e.g. user profiles, posts, etc.)
- **Streaming APIs**: Changes in users' data in real time (e.g. new tweets, weather alerts. . .)

APIs generally have extensive documentation:

- Written for developers who are building apps or websites
- What to look for: **endpoints** and **parameters**

Endpoints: web location that receives requests & sends responses

Parameters: allows you to send (very) specific requests

Endpoint + parameters = customised URL for making a request

Authentication and usage tracking

Many APIs require a **key** or **tokens** (often generated via a developer account)

Most APIs are **rate-limited**:

- Restrictions on number of **API calls** by user/key/IP address and period of time
- Commercial APIs may impose a monthly fee (via your account)

Just because APIs allow access, doesn't give you *carte blanche*:

- Commercial and non-commercial APIs typically have terms of use
- Read what you can/cannot do with the API and the data!

An Access Example: The NYTimes API

The New York Times (NYT) offers a range of APIs

In your seminar you will use:

- The **Article Search API** to search for keywords in articles
- The **Archive API** to download the full data for a given month

These public APIs do not give full article access, but they do give headlines, abstracts, snippets, and/or lead paragraphs since 1851

You will need to set up an account (i.e., before seminar):

- Follow these instructions:
<https://developer.nytimes.com/get-started>
- When specifying access rights, select the boxes for Article Search API and Archive API
- The system will generate a **private** key for access

Digression on digital security

You'll need to set up an API key in order to make calls to many APIs

→ Your key helps them track your usage...

A key is a password: securing it is **your responsibility**

→ If you don't, other people may use your account at your expense

Since we will be making API calls in R, you'll need to get your API keys into R somehow

Never ever hard-code an API key (or *any* password) into a script that another person/developer/user will see, e.g.:

```
my_password <- "my password for everyone to see!"
```

Digression on digital security

If you haven't already, you should spend some time securing your digital life and setting up a digital security “system”

This would include:

- storing credentials in a password manager
- setting up dual authentication on all your accounts
- using **unique** and **complex** passwords for every account
- using passkeys where possible
- never storing passwords or other sensitive information in unsecured locations (e.g., unprotected text files or loose papers)

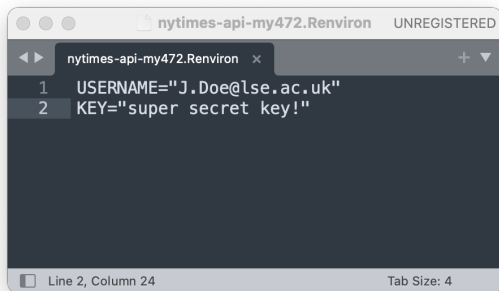
More generally: where possible, encrypt your data and be weary about storing personal information in the cloud

Storing credentials such as your NYT API key

As mentioned, you shouldn't hard code your passwords

Old way of dealing with this was `.Renviron` (or just `.env`) files

- Create plain text file, saved **locally** with `.Renviron` file suffix
- Each line should have information you want to store, formatted in a specific way, e.g.:



A screenshot of a code editor window titled "nytimes-api-my472.Renviron" with a status bar indicating "UNREGISTERED". The editor shows two lines of code: "1 USERNAME='J.Doe@lse.ac.uk'" and "2 KEY='super secret key!'". The status bar at the bottom indicates "Line 2, Column 24" and "Tab Size: 4".

```
nytimes-api-my472.Renviron UNREGISTERED
1 USERNAME='J.Doe@lse.ac.uk'
2 KEY='super secret key!'
Line 2, Column 24 Tab Size: 4
```

Storing credentials such as your NYT API key

Then, when you need a key or other piece of info:

→ Read the environment details into R:

```
readRenviron("nytimes-api-my472.Renviron")
```

→ Then get whatever piece of info you need:

```
Sys.getenv("KEY")
```

To emphasise: if done correctly, this will only work on *your* computer, since you won't share your `.Renviron` file with anyone!

Never push an `.Renviron` file to GitHub!

Storing credentials such as your NYT API key

Recall: I said you shouldn't store credentials in unsecure files!

→ An `.Renviron` file is an unsecure file...

There is now a better way to do handle keys: store your credentials in your computer's "keychain" using `keyring` package

→ In macOS the keychain is called "Keychain Access"

→ In Windows the keychain is called "Credential Manager"

```
## Installation -- do this once (if needed)
# install.packages("pak")
# pak::pak("keyring")
library(keyring)
key_set("nyt-api-key") # add a key to keychain
key_get("nyt-api-key") # get a key from keychain
```

JSON

JSON

- API responses are very often delivered in JSON format (JavaScript Object Notation)
- JSON is a lightweight, flexible, easy-to-parse format to store and transmit data
- JSON data can be read/parsed into R with the `fromJSON` function from the `jsonlite` package
- Yet, many packages have their own functions to read data in JSON format into R, e.g. the `content(r, ...)` function from the `httr` package which we will use a little later

JSON

- JSON objects are key-value pairs: `"someKey": [someValue]`
- Many key-value pairs can be in a single JSON object, separated with `", "`
- Keys have to be strings with double quotes
- Values can be one of the following types:
 - String (e.g., `"hello"`)
 - Number (e.g., `42`, `3.141`)
 - Array (e.g., `["a", "b", "c"]`)
 - Boolean (e.g., `true`, `false`)
 - `null`
- Often follow a **nested** structure
- Can be represented as R `list()` (or python `dict()`)

Reference: https://www.w3schools.com/js/js_json_syntax.asp

JSON example (adapted from Wikipedia)

```
{
  "USER261728": {
    "first_name": "John",
    "last_name": "Smith",
    "is_alive": true,
    "age": 27,
    "address": {
      "street_address": "21 2nd Street",
      "city": "New York",
      "state": "NY",
      "postal_code": "10021-3100"
    },
    "children": [
      "Catherine",
      "Thomas",
      "Trevor"
    ],
    "spouse": null
  }
}
```

Coding

→ 01-json-in-r.Rmd

API calls

Constructing an API call with Google Maps API

As with all APIs, start by reviewing the documentation:

<https://developers.google.com/maps/documentation/geocoding>

To construct an API call, you need:

- Endpoint: `https://maps.googleapis.com/maps/api/geocode/json`
- Parameter(s) of call: `address=london`
- Authentication token: `key=XXXXXX` [technically a parameter]

With this information, you can construct a URL that gives instructions to the API to return the data you want

- Use `?` to start specifying parameters
- Separate parameters with `&`
- Replace spaces with `%20`

`https://maps.googleapis.com/maps/api/geocode/json?address=london&key=XXXXXX`

Constructing an API call with Google Maps API

If you navigate to this url, you'll see the data in your browser

→ But we want it in R!

Since this endpoint will produce a JSON file (read the docs!) we can use the `jsonlite` package as follows

```
library(tidyverse)
library(jsonlite)
library(keyring)

url <- "https://maps.googleapis.com/maps/api/geocode/json" %>%
  paste0("?address=london&key=") %>%
  paste0(key_get("google-maps-api-key"))

r <- fromJSON(url)

print(r$status) # "OK" means request was successful!
```

Constructing an API call with Google Maps API

There is another way to construct and make an API call using the `httr` package

```
library(httr)
url <- "https://maps.googleapis.com/maps/api/geocode/json"

r <- GET(url, query=list(address="london",
                        key=key_get("google-maps-api-key")))

print(r$status_code) # "200" means request was successful!
```

(Status code 4xx are client errors; 5xx are server errors)

GET creates the API URL for your call and requests it

Advantage of `httr`: you can also *attach* data using a POST request

➔ But we won't do this in MY472

Constructing an API call with Google Maps API

```
{
  "results" : [
    {
      "address_components" : [
        {
          "long_name" : "London",
          "short_name" : "London",
          "types" : [ "locality", "political" ]
        },
        {
          "long_name" : "London",
          "short_name" : "London",
          "types" : [ "postal_town" ]
        }
      ],
      ...
    }
  ]
}
```

Constructing an API call with Google Maps API

```
...  
    "formatted_address" : "London, UK",  
    "geometry" : {  
      "bounds" : {  
        "northeast" : {  
          "lat" : 51.6723432,  
          "lng" : 0.148271  
        },  
        "southwest" : {  
          "lat" : 51.384940099999999,  
          "lng" : -0.3514683  
        }  
      },  
      "location" : {  
        "lat" : 51.5073509,  
        "lng" : -0.1277583  
      },  
      "location_type" : "APPROXIMATE",  
    }  
  }  
}
```

Constructing an API call with Google Maps API

```
...  
    "viewport" : {  
        "northeast" : {  
            "lat" : 51.6723432,  
            "lng" : 0.148271  
        },  
        "southwest" : {  
            "lat" : 51.384940099999999,  
            "lng" : -0.3514683  
        }  
    },  
    "place_id" : "ChIJdd4hrwug2EcRmSrV3Vo6lllI",  
    "types" : [ "locality", "political" ]  
  },  
  ],  
  "status" : "OK"  
}
```

Constructing an API call with Google Maps API

```
# extract the content  
r_content <- content(r, as="parsed")
```

```
# how many results?  
length(r_content$results)
```

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

```
# get some data from the first result  
r_content$results[[1]]$formatted_address
```

```
## [1] "London, UK"
```

```
r_content$results[[1]]$geometry$location$lat
```

```
## [1] 51.50722
```

```
r_content$results[[1]]$geometry$location$lng
```

```
## [1] -0.1275862
```

Coding

→ 02-aic-api.Rmd