THE LONDON SCHOOL OF ECONOMICS AND POLITICAL SCIENCE

Week 5: Accessing Data in the Cloud

MY472: Data for Data Scientists
https://lse-my472.github.io/

Autumn Term 2025

Ryan Hübert
Associate Professor of Methodology

Slides last updated: 29 October 2025

Outline

Getting data

The internet contains *a lot* of data

Much of it is relevant for social scientists

- Archived datasets
- Administrative data
- Social media posts
- Speeches and legal documents
- News articles

You'll inevitably want to get data from the web

Ways to get data

Getting data is hard, in general

In this class so far:

- ▷ We gave you data (via the course website/GitHub)
- ▷ You got data from websites using `download.file()`
  - A programmatic alternative to clicking to download

Data is available via many more online channels

- ▷ Stored in cloud databases
- ▷ Retrievable through APIs
- ▷ Embedded into webpages

These ways of getting data differ in ease of access, amount of processing required, ethical considerations

Getting data from the cloud

This week:

- big picture stuff (ethics, digital data security)
- "easy" online data collection: APIs

Weeks 7 and 8:

- "harder" online data collection: static webscraping
- "hardest" online data collection: dynamic webscraping

Outline

The two core ideas

Core idea 1: Not all available data is meant to be used for data science

Core idea 2: You need to (learn how to) exercise good judgement

Today: some key principles relevant for what we're doing here, but there is much more to research ethics

Some ethical considerations

Research ethics is a somewhat fuzzy concept, but most people think it relates to two broad questions:

- What is *legal*?
- What is *morally acceptable*?

There are several ethical considerations relating to online data collection, which touch on legality and moral acceptability, e.g.:

- Who owns the data?
- What permissions have been given?
- Does data contain personally identifying or sensitive data?
- Is the data vulnerable to theft or leaks?

Evaluating ethical considerations

How you evaluate ethical considerations depends on your context:

- Academic: stringent internal norms, but more legal flexibility
- Industry: laxer internal norms, but more legal constraints
- Government: often quite unclear

What could go wrong?

- You could get disciplined by LSE, expelled, have degree revoked
- You could get fired from a job, or have your research publicly retracted or criticised
- You could get sued in a court
- You could get criminally investigated or prosecuted

Online data collection rules of thumb

For online data collection, some useful rules of thumb:

1. Just because you can see something in a browser does not mean you should copy it elsewhere
2. Webservers distinguish between users: bots versus browsers versus authenticated users
3. You should do your best to obey a webserver's terms of service, its `robots.txt` and its server behaviour
4. When possible, tell a webserver who you are
5. Be considerate about resource use
6. If a webserver "offers" data you need (via an API or to download), get it that way
7. Be proactive (and aggressive) about securing data

Outline

Why should you care about digital security?

Many, many reasons to care about digital security, e.g.

- Malicious actors are trying to steal your money and/or your data
- (Some) governments may be trying to spy on you
- You should protect yourself against unintended billing
- You may need to handle, store and protect sensitive data in your future job

You should probably be more worried about this than you are

Digital security is an ethics issue

Ethical obligations extend to data security

- You must take *proactive* steps to prevent unauthorised access to your resources and data

This means you need to be serious about:

- Authentication: nobody should be able to access your devices or accounts
- Secure storage: if there *is* unauthorised access, nobody should be able to make sense of your data

We'll cover authentication now, as it is relevant for accessing APIs

We'll cover secure storage in a later week

Authentication approaches

Authentication is the process by which your identity is verified via **credentials** and you are given access to a resource

- You authenticate to use your devices
- You authenticate to use cloud-based services

The most basic way to authenticate: provide two credentials, a username and a password

Passwords are still a core part of authentication

- Use complex passwords
- Do not reuse passwords
- Do not share passwords
- Do not store passwords in unprotected files

Authentication approaches

What is a “complex” password? Two good approaches:

1. Random password mixing character types, like:

`u__akWTb82UAkt94FMPxhX2voccd3Znu7Be.ynHVN4u_ZVfE9E`

2. “Memorable” multi-word phrase, like

`southern ridge leased suites peter`

Combining both approaches is even better:

`$outhern Ridge l3ased su!tes`

No matter what, should be sufficiently long

Authentication approaches

Most cloud services now use **multifactor authentication (MFA)**

- The idea: multiple forms of authentication is better than one
- SMS codes are less secure than authenticator apps
- But sending passwords over internet is still risky

A newer approach avoids passwords: **passkeys**

- Device creates passkey by generating pair of keys: private key kept on device and public key stored by the service
- Authentication: unlock private key with biometrics or PIN
- Private key signs a “challenge” from the service, and signature is verified using the public key
- Very secure, but unfortunately still buggy across devices

Authentication approaches

In more programmatic settings, you will often need **tokens** or **keys** to access resources from a webserver

- Machine-friendly forms of credentials, proving to a server who you are and what you’re allowed to do
- E.g., GitHub has *personal access tokens* and *SSH keys*

Designed for “developers”: anyone whose program needs to talk to another service securely

A distinction (although somewhat fuzzy):

- “Passwords” authenticate real humans
- “Tokens” and “keys” authenticate a *program* acting on behalf of human in an automated way

Storing and accessing credentials

Many ways to store credentials

Some very bad ways:

- Trying to remember them
  - This incentivises you to use insecure passwords, and to reuse passwords (or make only slight alterations)
  - You simply cannot do this for developer tokens/keys
- Storing in unsecured files like `MyPasswords.docx`
  - Anyone can access all your passwords if they get into your computer
  - Easy to mistakenly expose (e.g., push to GitHub)

Storing and accessing credentials

Use a **password manager** for your passwords

- Password managers store passwords and auto-fill when needed (only after device authentication)
- Password managers help you create very secure passwords
- Password managers can do *a lot* more than this: reminders to reset passwords, MFA tokens, etc.

Modern operating systems and many browsers have password managers built in

- Newest macOS/iOS versions have “Passwords” app
- Google Chrome has “Google Password Manager”

But also many apps you can buy: 1Password, LastPass, etc.

Storing and accessing credentials

In development settings, you'll need programmatic access to credentials

- For example: many APIs require you to have a key to authenticate (so they can track use and/or bill you)
- For some tasks in MY472, you will need to access keys in R

Never ever hard-code a key/token/password, e.g.:

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

Storing and accessing credentials

Older way of storing credentials for use in R: `.Renviron` files

- Create plain text file, saved locally with `.Renviron` file suffix
- Can also use `.env` file extension for brevity
- Each line should have information you want to store, formatted in a specific way, e.g.:

key.Renviron

```
USERNAME="J.Smith@lse.ac.uk"
KEY="super secret key!"
```

Storing and accessing credentials

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

1. Read the environment details into R:

```r
readRenviron("key.Renviron")
```

2. Get whatever piece of info you need:

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

```
[1] "super secret key!"
```

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 and accessing credentials

Recall: you shouldn't store credentials in unsecure files

- And an `.Renviron` file is an unsecure file!

A more secure way to handle credentials: use your computer's "keychain"

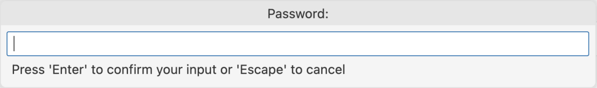
- A **keychain** is a piece of operating system software that stores sensitive information (like tokens) for use by other software
- This is not the same thing as a password manager, which is meant for human (not software) use
- In macOS the keychain is called "Keychain Access", in Windows the keychain is called "Credential Manager"

Can access keychain in R using `{keyring}`

Storing and accessing credentials

```r
library("keyring")
key_set("example-key") # add a key to keychain
```

In Positron, a pop-up window will appear at the (very) top, asking you to enter the token you want to add to the keychain:



You should enter your token in this box and then press enter

Can also save tokens in a password manager as back up

You can manually edit tokens directly in the keychain app

Storing and accessing credentials

Then, when you need a token from your keychain:

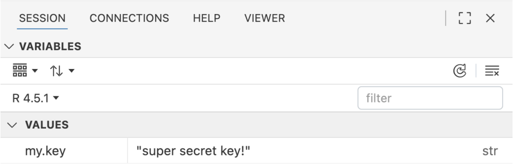
```r
my.key <- key_get("example-key") # get a key from keychain
```

Token is not visible in R console/editor, unless it's printed:

```r
print(my.key) # oops! I revealed my token!
```

```
[1] "super secret key!"
```

But, it *is* visible in Positron's variables pane (yikes!):

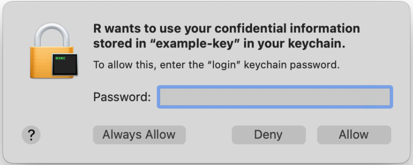


Storing and accessing credentials

When you use the `{keyring}` package, you might get a security message from your OS

It will ask you to approve R's access to your OS keychain

For example, on macOS, this may pop up:



Will need to enter your computer's (admin) password to allow it

Advice to protect your accounts

1. Spend time securing your devices, accounts

- ▷ use aggressive settings to protect your devices, including password unlock, biometric, loss/theft settings
- ▷ use unique and complex passwords for all accounts
- ▷ set up MFA on all your accounts
- ▷ store passwords in a password manager, tokens in keychain
- ▷ use passkeys where possible
- ▷ never store sensitive information in unsecured locations (e.g., unprotected text files or loose papers)

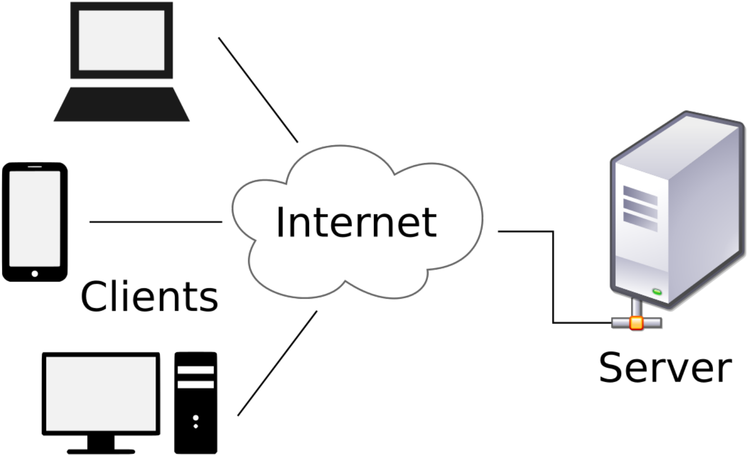
Advice to protect your accounts

2. Be suspicious

- Do not give passwords/tokens to anyone, seriously: anyone
- Do not reply to messages from people you do not know
  - Keep your contacts up to date and synced across devices
- Carefully check email addresses and URLs
  - Hover before clicking
  - Do not click on weird links from random internet people
  - Look for `https` on websites
- Do not click past warnings from your browser or device
- Disable remote image loading in your email account

Outline

Client-server model



Client-server model

Client: User computer, tablet, phone, software application, etc.

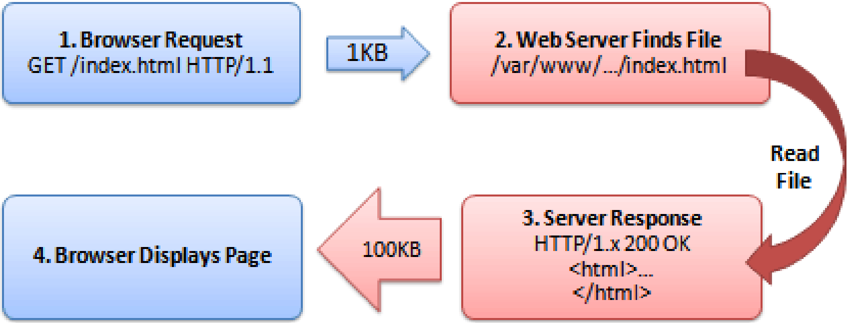
Server: Web server, mail server, file server, etc.

In **request–response systems**:

- ▹ Client makes **request** to the server
- ▹ Depending on what you want to get, the request might be
  - **HTTP: Hypertext Transfer Protocol**
  - **HTTPS: Hypertext Transfer Protocol Secure**
  - **SMTP: Simple Mail Transfer Protocol**
  - **FTP: File Transfer Protocol**
- ▹ Server returns **response**

Example request and response for HTTP

From StackOverflow



Anatomy of a client request

A client sends a request with several components:

- **URL**: where the request is sent—a remote “path”
- **Method**: what kind of action you’re asking for
  - `GET` to retrieve (digital) data
  - `POST` to send or submit data
  - `PUT` or `DELETE` to modify or remove data
- **Headers**: metadata about the request represented as key-value pairs, e.g.:
  - **User-Agent**: who/what is making request
  - **Authorization**: provides credentials like tokens
- **Body**: data being sent (only for `POST`/`PUT` requests)

Anatomy of a server response

A server sends back a response with several components

- **Status code**: was request fulfilled? For example: `200 OK`, `403 Forbidden`, or `404 Not Found`
  - Status code 4xx are client errors; 5xx are server errors
- **Headers**: info about the response, for example
  - **Content-Type**: what kind of data
  - **Content-Length**: length in bytes
  - **RateLimit-Remaining**: how many requests before throttling
- **Body**: the content requested, such as
  - HTML if you're accessing a webpage
  - JSON or XML if you're accessing an API

Browsing the web in R

The {httr} package allows you to make requests in R

```r
library("httr")
response <- GET(url = "https://lse-my472.github.io/")
print(response)
```

```
Response [https://lse-my472.github.io/]
  Date: 2025-10-26 14:59
  Status: 200
  Content-Type: text/html; charset=utf-8
  Size: 37.5 kB
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en"><head>

<meta charset="utf-8">
<meta name="generator" content="quarto-1.7.32">

<meta name="viewport" content="width=device-width, initial-scale=1.0, user-s


<title>MY472: Data for Data Scientists – MY472</title>
...
```

Browsing the web in R

```
response$status_code
```

```
[1] 200
```

Content should be a webpage (coded in HTML), but it’s returned in raw bytes:

```
response$content[1:20] # first 20 bytes
```

```
[1] 3c 21 44 4f 43 54 59 50 45 20 68 74 6d 6c 3e 0a 3c 68 74 6d
```

Can get properly encoded text as we saw in week 2:

```
rawToChar(response$content[1:20])
```

```
[1] "<!DOCTYPE html>\n<htm"
```

Can also extract the text of the response using `content(response, "text")` from `{httr}`

User-Agent header

User-agent header identifies software making a request, so that:

- ▷ Different content goes to different devices (desktop v. mobile)
- ▷ Servers can detect and manage bots or crawlers
- ▷ Servers can block suspicious or aggressive automated access

{httr} includes a default user-agent in requests:

```
response$request
```

```
<request>
GET https://lse-my472.github.io/
Output: write_memory
Options:
* useragent: libcurl/8.11.1 r-curl/6.2.2 httr/1.4.7
* httpget: TRUE
Headers:
* Accept: application/json, text/xml, application/xml, */*
```

User-Agent header

It's good practice to identify yourself clearly

- This is a key part of ethical data collection from the web

You can set a user-agent with `{httr}`, e.g.:

```
ua <- "LSE-MY472/2025 (educational use; contact: USERNAME@lse.ac.uk)"
response <- GET(url = "https://lse-my472.github.io/", user_agent(ua))
```

This tells the site administrator of the webserver who you are and how to contact you if your script causes issues

- Use *your* email address!

Some webservers have preferred formats—check!

Limit resource use

When you make requests, you are using the server's resources

It is good practice to minimise your impact on the webserver

- ▷ This is another key part of ethical data collection from the web

When you make requests, especially if you make several in a row, you should include a delay:

```
response <- GET(url = "https://lse-my472.github.io/", user_agent(ua))
Sys.sleep(0.5) # this is in seconds
```

You must use good judgement

- ▷ Set reasonable delays depending number of requests
- ▷ Embed delays at *correct* locations in code (after requests!)

Outline

APIs

Application programming interface (API)

- defined set of functions, classes, or methods
- allows one piece of software interact with another
- work without needing to understand its internal workings

Key points:

- APIs are not user interfaces, though they can power one
- Used to connect and exchange data between applications

As an example: every R package is an API

Web APIs

Web APIs: interfaces to interact with remote servers

- **RESTful APIs**: structured HTTP/S requests and responses
  - Good for static information, e.g. user profiles, posts, etc.
  - Often return data in lightweight formats like JSON or XML
  - More broadly, they're used to retrieve, create, update, or manage data and user accounts on remote servers
- **Streaming APIs**: persistent connections with exchange of real time information

We'll focus on RESTful APIs in this class

Working with web APIs

There are many, many RESTful APIs available, for example:

- Google (Maps, Gmail, etc.)
- Microsoft 365
- Dropbox
- *New York Times*
- HMRC

APIs generally have extensive documentation:

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

Working with web APIs

Endpoints: web location for requests & responses

Parameters: allows you to send customised requests

- Endpoint + parameters = customised URL for making a request

Many APIs require a key or tokens, and 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)
- Be sure to follow terms of service

API keys/tokens can be supplied as a parameter (i.e., in the URL) or in an HTTP/S request header

Constructing an API call with Google Maps API

We'll access the Geocoding API, which requires authentication with a token

- A "simple" token: API doesn't touch user data
- Supplied as a parameter (in the API call URL)

To replicate, you will need a Google Cloud account

- Activate at `https://cloud.google.com/`
- Create a project
- In settings enable the Geocoding API, and get a key
- Store in keychain

```r
library("keyring")
key_set("google-maps-api-key")
```

Constructing an API call with Google Maps API

Docs: `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` and `key`

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`

Constructing an API call with Google Maps API

Suppose we want to look up geocoordinates for New York City

The URL we need looks something like this:

`https://maps.googleapis.com/maps/api/geocode/json?address=new%20york%20city&key=XXXXX`

Note: this url won't work! I have not included a real key

Constructing an API call with Google Maps API

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

- But we want the data in R!

To construct the API call, first specify the end point

```
endpoint <- "https://maps.googleapis.com/maps/api/geocode/json"
```

Then construct the URL, specifying parameters

```
library("keyring") # to get token
library("httr")    # to make requests

url <- endpoint |>
  paste0("?address=new%20york%20city") |>
  paste0("&key=")

r <- GET(url = paste0(url, key_get("google-maps-api-key")))
```

Constructing an API call with Google Maps API

Another way to construct a call using `{httr}`

- Do not write full URL manually
- Provide parameters in a list

```
r <- GET(url = endpoint, query = list(address="new york city",
                                     key=key_get("google-maps-api-key"))
```

```
print(r$status) # 200 is ok!
```

```
[1] 200
```

```
print(r$headers$`content-type`)
```

```
[1] "application/json; charset=UTF-8"
```

JSON

API responses are very often delivered in **JSON** format

- JSON stands for **J**ava**S**cript **O**bject **N**otation

JSON is a lightweight, flexible, easy-to-parse format to store and transmit data

- JSON objects consist of key-value pairs
- Many key-value pairs can be in a single JSON object, separated with comma
- Keys have to be strings with double quotes
- Values can be strings, numbers, arrays, booleans or `null`
- Can have a **nested** structure

JSON examples

This is data stored in JSON format

```json
{
  "USER261728": "John Smith",
  "USER261729": "Jane Doe"
}
```

Two key-value pairs:

- Keys are strings representing user IDs
- Values are strings representing full names of users

JSON examples


```json
{
  "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
  }
}
```


Adapted from Wikipedia

JSON in R

In R, JSON data can be read into R and parsed with the `fromJSON()` function from the `{jsonlite}` package

- Renders JSON as a `list` object in R

Many packages have their own functions to read data in JSON format into R

- The `{httr}` package has `content(r, ...)`

JSON in R

```r
library("jsonlite")
users <- fromJSON("data/simple_example.json")
print(users)
```

```
$USER261728
[1] "John Smith"

$USER261729
[1] "Jane Doe"
```

Can pull info out using R `list` syntax

```r
users$USER261729
```

```
[1] "Jane Doe"
```

Google Maps API result

Back to our Google Maps API result

```
prettify(rawToChar(r$content))
```


```
{
    "results": [
        {
            "address_components": [
                {
                    "long_name": "New York",
                    "short_name": "New York",
                    "types": ["locality", "political"]
                },
                {
                    "long_name": "New York",
                    "short_name": "NY",
                    "types": ["administrative_area_level_1", "political"]
                },
                {
                    "long_name": "United States",
                    "short_name": "US",
                    "types": ["country", "political"]
                }
            ],
            "formatted_address": "New York, NY, USA",
```

Google Maps API result


```
"geometry": {
    "bounds": {
        "northeast": {
            "lat": 40.917705,
            "lng": -73.700169
        },
        "southwest": {
            "lat": 40.476578,
            "lng": -74.258843
        }
    },
    "location": {
        "lat": 40.7127753,
        "lng": -74.0059728
    },
    "location_type": "APPROXIMATE",
    "viewport": {
        "northeast": {
            "lat": 40.917705,
            "lng": -73.700169
        },
        "southwest": {
            "lat": 40.476578,
            "lng": -74.258843
        }
    }
},
```

Google Maps API result


```
            "place_id": "ChIJOwg_06VPwokRYv534QaPC8g",
            "types": ["locality", "political"]
        }
    ],
    "status": "OK"
}
```

Google Maps API result

Let's use `content()` to parse JSON from `{httr}` response object

```
results <- content(r) # auto formats
str(results)
```

```
List of 2
 $ results:List of 1
  ..$ :List of 5
  .. ..$ address_components:List of 3
  .. .. ..$ :List of 3
  .. .. .. ..$ long_name : chr "New York"
  .. .. .. ..$ short_name: chr "New York"
  .. .. .. ..$ types     :List of 2
  .. .. .. .. ..$ : chr "locality"
  .. .. .. .. ..$ : chr "political"
[...rest cut off...]
```

Google Maps API result

```r
results$results[[1]]$formatted_address
```

```
[1] "New York, NY, USA"
```

```r
results$results[[1]]$geometry$location
```

```
$lat
[1] 40.71278

$lng
[1] -74.00597
```

JSON data can be stored locally in `.json` files

```r
write_json(results, path = "nyc_coords.json")
```