

# Introduction to Data Science

## Session 7: Web scraping and APIs

---

Simon Munzert

Hertie School | GRAD-C11/E1339

# Table of contents

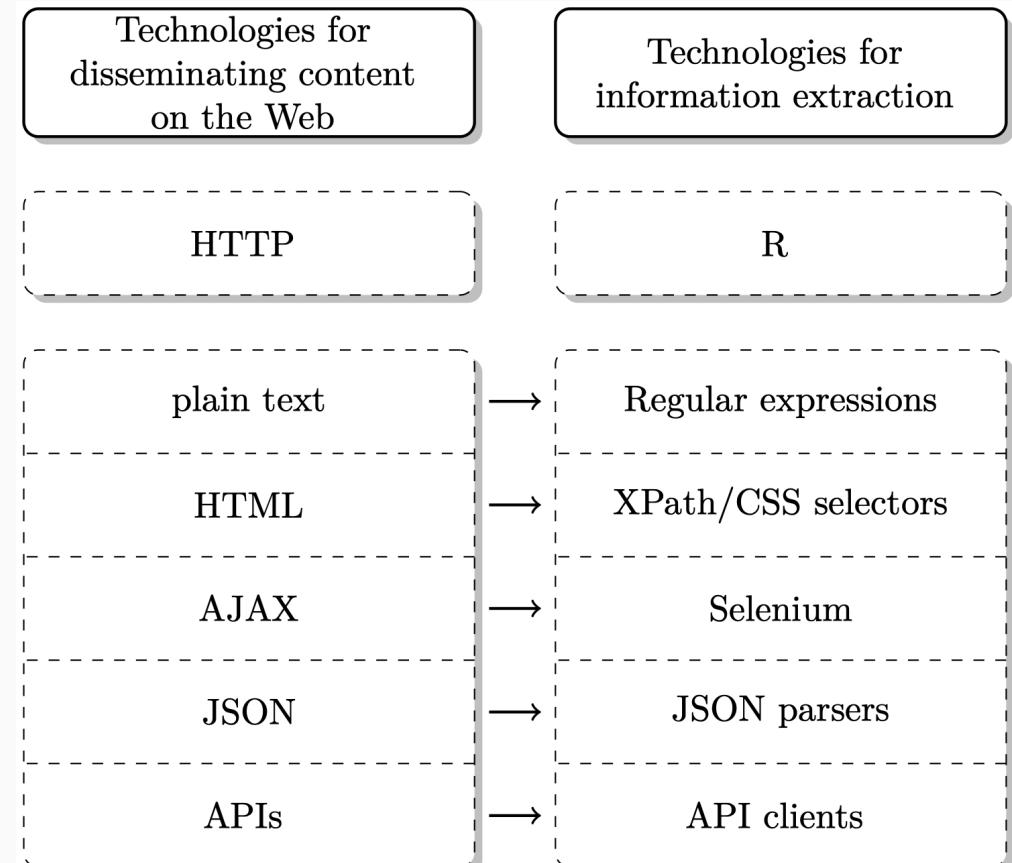
1. Scraping static webpages with R
2. Web scraping: good practice
3. Web APIs: the basics
4. JSON
5. Summary

# Scraping static webpages with R

---

# Technologies of the world wide web

- To fully unlock the potential of web data for data science, we draw on certain web technologies.
- Importantly, often a basic understanding of these technologies is sufficient as the focus is on web data collection, not **web development**.
- Specifically, we have to understand
  - How our machine/browser/R communicates with web servers (→ **HTTP/S**)
  - How websites are built (→ **HTML, CSS**, basics of **JavaScript**)
  - How content in webpages can be effectively located (→ **XPath, CSS selectors**)
  - How dynamic web applications are executed and tapped (→ **AJAX, Selenium**)
  - How data by web services is distributed and processed (→ **APIs, JSON, XML**)



Credit **ADCR**

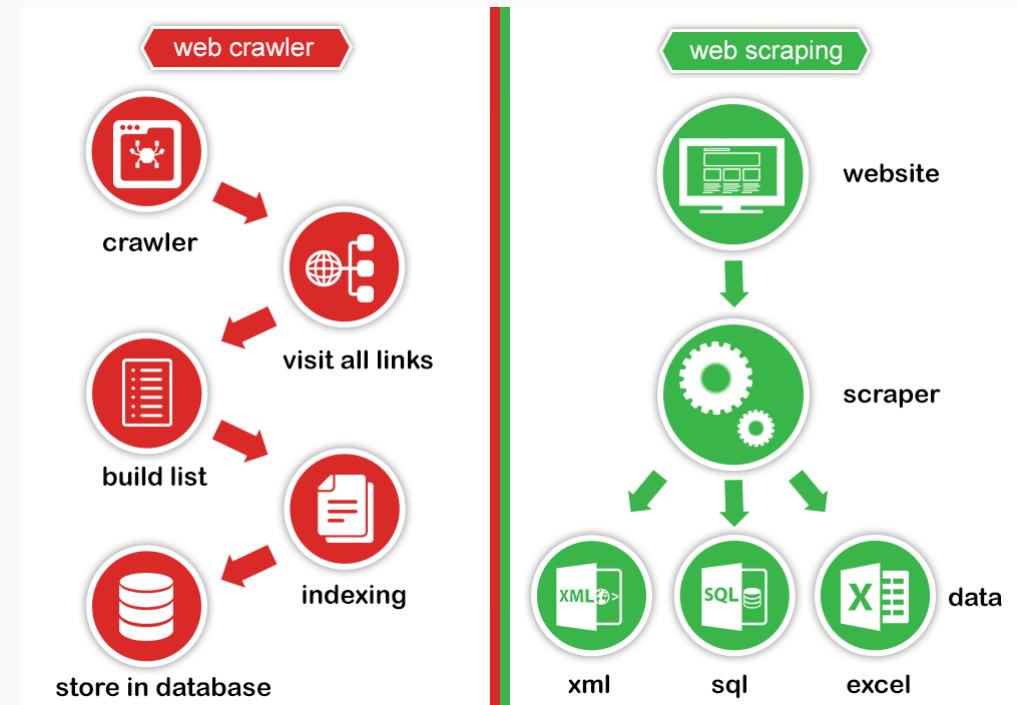
# Web scraping

## What is web scraping?

1. Pulling (unstructured) data from websites (HTMLs)
2. Bringing it into shape (into an analysis-ready format)

## The philosophy of scraping with R

- No point-and-click procedure
- Script the entire process from start to finish
- **Automate**
  - The downloading of files
  - The scraping of information from web sites
  - Tapping APIs
  - Parsing of web content
  - Data tidying, text data processing
- Easily scale up scraping procedures
- Scheduling of scraping tasks



Credit [proweb scraping.com](http://proweb scraping.com)

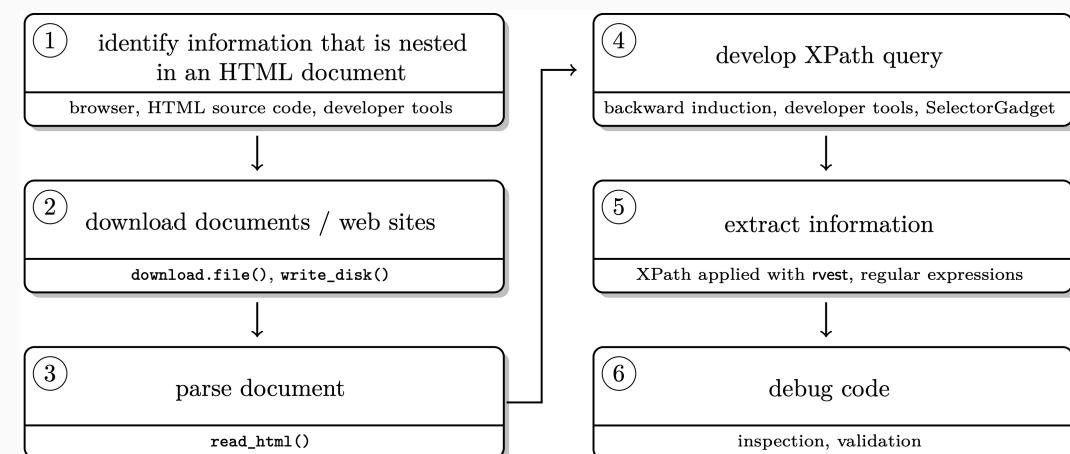
# The scraping workflow

## Key tools for scraping static webpages

1. You are able to inspect HTML pages in your browser using the web developer tools.
2. You are able to parse HTML into R with `rvest`.
3. You are able to speak XPath (or CSS selectors).
4. You are able to apply XPath expressions with `rvest`.
5. You are able to tidy web data with R/ `dplyr` / `regex`.

## The big picture

- Every scraping project is different, but the coding pipeline is fundamentally similar.
- The (technically) hardest steps are location (XPath, CSS selectors) and extraction (clean-up), sometimes the scaling (from one to multiple sources).



# Web scraping with rvest

`rvest` is a suite of scraping tools. It is part of the tidyverse and has made scraping with R much more convenient.

There are three key `rvest` verbs that you need to learn.<sup>1</sup>

1. `read_html()`: Read (parsing) an HTML resource.
2. `html_elements()`: Find elements that match a CSS selector or XPath expression.
3. `html_text2()`: Extract the text/value inside the node set.



<sup>1</sup> There is more in `rvest` than what we can cover today. Have a glimpse at the [overview at tidyverse.org](#) and at this excellent (unofficial) [cheat sheet](#).

# Web scraping with rvest: example

- We are going to scrape information from a Wikipedia article on women philosophers available at

[https://en.wikipedia.org/wiki/  
List\\_of\\_women\\_philosophers.](https://en.wikipedia.org/wiki/List_of_women_philosophers)

- The article provides two types of lists - one by period and one sorted alphabetically. We want the alphabetical list.
- The information we are actually interested in - names - is stored in unordered list elements.

This screenshot shows the Wikipedia page for 'List of women philosophers'. The sidebar on the left contains links for Main page, Contents, Recent events, Recent changes, About Wikipedia, Contact us, Donate, Help, Learn to edit, Community portal, Recent changes, Upload file, Tools, What links here, Related changes, Special pages, Permanent link, Page protection, Cite this page, Wikipedia talk, Print/export, Download as PDF, Printable version, Languages, Deutsch, Español, Magyar, Português, Svenska, 中文, and Edit links. The main content area displays a list of women philosophers, starting with Anne Conway, Mary Wollstonecraft, Mary Murdoch, Elizabeth Anscombe, Joyce Michel Cook, Coretta Scott King, and others, followed by a section on 'Controversy (edit)'.

This screenshot shows the same Wikipedia page, but the list has been sorted alphabetically. It includes a header 'Alphabetically [edit]' and a 'Contents' section with links for A-B-C-D-E-F-G-H-I-J-L-M-N-O-P-Q-R-S-T-U-V-W-X-Y-Z. The main list contains names like Felicia Nimue Ackerman, Martha Kiese, Patricia Hitchcock, Eva Hartmann, Karen Keck, Martha Kwein, Helen Knight, Sarah Kofman, Christine Kongsgaard, Julia Kristeva, and Inez Kugund. There is also a small portrait of Tullia d'Aragona.

This screenshot shows the same Wikipedia page, but the list has been sorted by period. It includes a header 'By period [edit]' and a 'Contents' section with links for Ancient philosophy, Medieval philosophy, Renaissance philosophy, Enlightenment philosophy, 19th century philosophy, 20th century philosophy, and 21st century philosophy. The main list contains names like Lopaputta, Mahayana, Ghotha, Gargi, Vachaspati, Hypatia of Alexandria, Arete, Simone de Beauvoir, Iris Murdoch, Elizabeth Anscombe, Mary Midgley, Ayn Rand, Hannah Arendt, Simone Weil, Iris Murdoch, Iris Marion Young, Joyce Michel Cook, and others, followed by a section on 'Ancient philosophy'.

```
> <h2>...</h2>
> <div class="noprint">...</div>
> <h3>...</h3>
> <style data-mw-deduplicate="TemplateStyles:r998391716">
> </style>
> <div class="div-col" style="column-width: 30em;">
>   <ul>
>     <li> == $0
>       :marker
>       <a href="/wiki/Felicia_Nimue_Ackerman" title="Felicia Nimue Ackerman">Felicia Nimue Ackerman</a>
>       " (fl. 2014)"
>     </li>
>     <li>...</li>
>     <li>...</li>
>     <li>...</li>
```

# Scraping HTML tables: example (cont.)

## Step 1: Parse the page

```
R> url_p <- read_html("https://en.wikipedia.org/wiki/List_of_women_philosophers")
```

## Step 2: Develop an XPath expression (or multiple) that select the information of interest and apply it

```
R> elements_set <- html_elements(url_p, xpath = "//h2/span[text()='Alphabetically']//following::li/a[1]")
```

The XPath expression reads:

- `//h2`: Look for `h2` elements anywhere in the document.
- `/span[text()='Alphabetically']`: Within that element look for `span` elements with the content "Alphabetically".
- `//following::li`: In the DOM tree following that element (at any level), look for `li` elements.
- `/a[1]` within these elements look for the first `a` element you can find.

# Scraping HTML tables: example (cont.)

## Step 3: Extract information and clean it up

```
R> phil_names ← elements_set %>% html_text2()  
R> phil_names[c(1:2, 101:102)]  
  
## [1] "A"           "B"           "Elisabeth of Bohemia"  
## [4] "Dorothy Emmet"
```

## Step 4: Clean up (here: select the subset of links we care about)

```
R> names_iffer ←  
+   seq_along(phil_names) ≥ seq_along(phil_names)[str_detect(phil_names, "Felicia Nimue Ackerman")] &  
+   seq_along(phil_names) ≤ seq_along(phil_names)[str_detect(phil_names, "Alenka Zupančič")]  
R> philosopher_names_clean ← phil_names[names_iffer]  
R> length(philosopher_names_clean)  
  
## [1] 267  
  
R> philosopher_names_clean[1:5]  
  
## [1] "Felicia Nimue Ackerman" "Marilyn McCord Adams"    "Aedesia"  
## [4] "Alia Al-Saji"          "Lilli Alanen"
```

# Quick-n-dirty static webscraping with SelectorGadget

## The hassle with XPath

- The most cumbersome part of web scraping (data tidying aside) is the construction of XPath expressions that match the components of a page you want to extract.
- It will take a couple of scraping projects until you'll truly have mastered XPath.

## A much-appreciated helper

- **SelectorGadget** is a JavaScript browser plugin that constructs XPath statements (or CSS selectors) via a point-and-click approach.
- It is available here: <http://selectorgadget.com/> (there is also a Chrome extension).
- The tool is magic and you will love it.

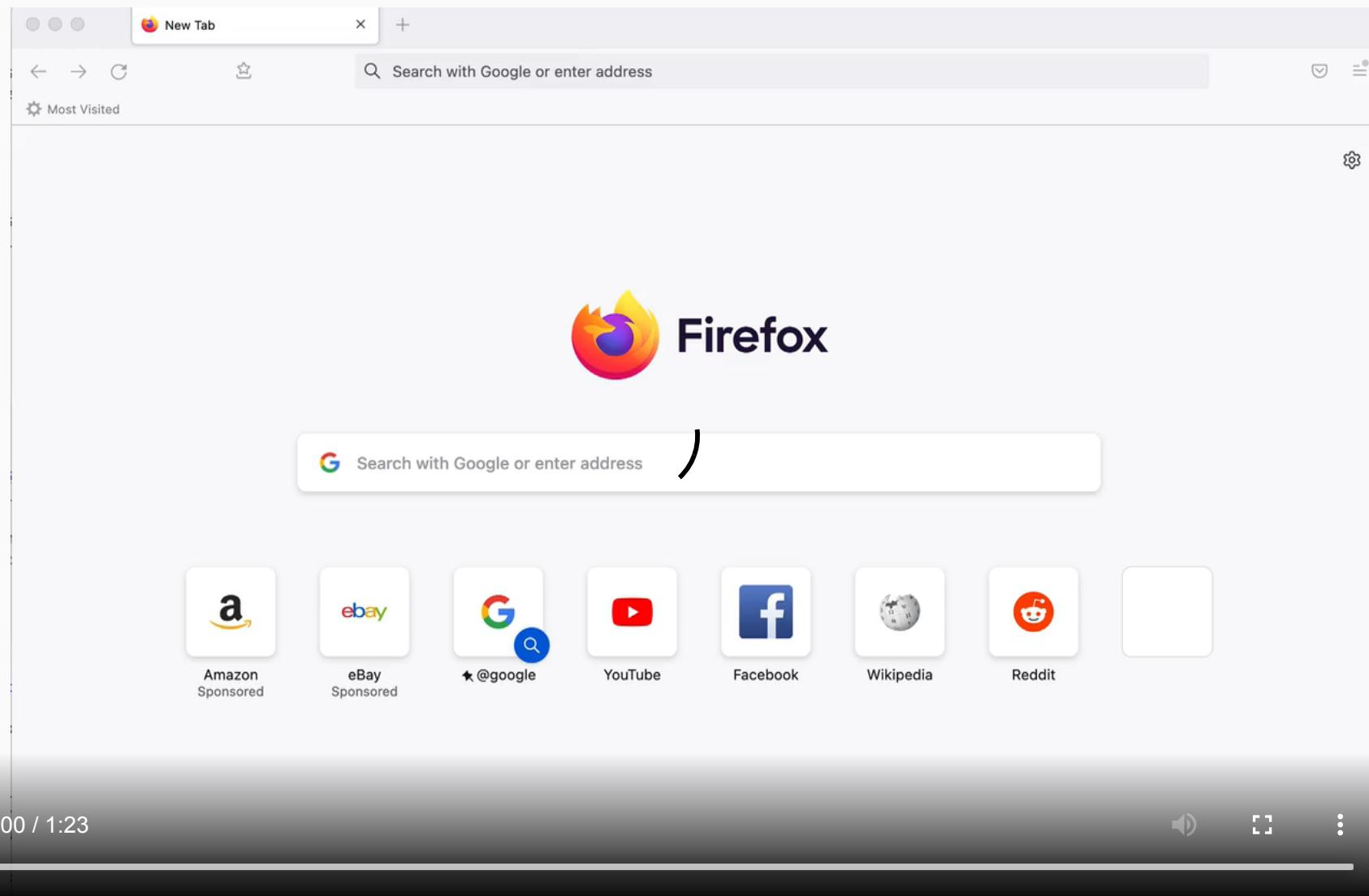
## What does SelectorGadget do?

- You activate the tool on any webpage you want to scrape.
- Based on your selection of components, the tool learns about your desired components and generates an XPath expression (or CSS selector) for you.

## Under the hood

- Based on your selection(s), the tool looks for similar elements on the page.
- The underlying algorithm, which draws on Google's diff-match-patch libraries, focuses on CSS characteristics, such as tag names and `<div>` and `<span>` attributes.

# SelectorGadget: example



# SelectorGadget: example (cont.)

```
R> library(rvest)
R> url_p <- read_html("https://www.nytimes.com")
R> # xpath: paste the expression from SelectorGadget!
R> # note: we use single quotation marks here (' instead of ") to wrap around the expression!
R> xpath <- '//*[@contains(concat( " ", @class, " " ), concat( " ", "erslblw0", " " ))]//*[@contains(concat( " ", 
R> headlines <- html_elements(url_p, xpath = xpath)
R> headlines_raw <- html_text(headlines)
R> length(headlines_raw)
R> head(headlines_raw)

## [1] 29

## [1] "Retailers' Latest Headache: Shutdowns at Their Vietnamese SuppliersRetailers' Latest Headache: Shutdowns at T
## [2] "With virus restrictions waning, it's becoming clear: Britain's gas crisis is a Brexit crisis, too. Here's why
## [3] "Business updates: U.S. stock futures signaled a rebound as bond yields fell back."
## [4] "Republicans at Odds Over Infrastructure Bill as Vote ApproachesRepublicans at Odds Over Infrastructure Bill a
## [5] "Liberals Dig In Against Infrastructure Bill as Party Divisions Persist"
## [6] "Successful programs from around the world could guide Congress in designing a paid family leave plan."
```

# SelectorGadget: when to use and not to use it

Having learned about a semi-automated approach to generating XPath expressions, you might ask:

## **Why bother with learning XPath at all?**

Well...

- SelectorGadget is not perfect. Sometimes, the algorithm will fail.
- Starting from a different element sometimes (but not always!) helps.
- Often the generated expressions are unnecessarily complex and therefore difficult to debug.
- In my experience, SelectorGadget works 50-60% of the times when scraping from static webpages.
- You are also prepared for the remaining 40-50%!

# Scraping HTML tables

Purchased Equipments (June, 2006)			
Item Num#	Item Picture	Item Description	Price
		Shipping Handling, Installation, etc	Expense
1.		IBM Clone Computer.	\$ 400.00
		Shipping Handling, Installation, etc	\$ 20.00
2.		1GB RAM Module for Computer.	\$ 50.00
		Shipping Handling, Installation, etc	\$ 14.00

Purchased Equipments (June, 2006)

Built	Building	City	Country	Roof	Floors	Pinnacle	Current status
1870	Equitable Life Building	New York City	 United States	043 m	142 ft	8	Destroyed by fire in 1912
1889	Auditorium Building	Chicago		082 m	269 ft	17	Standing
1890	New York World Building	New York City		094 m	309 ft	20	106 m 349 ft Demolished in 1955
1894	Philadelphia City Hall	Philadelphia		155.8 m	511 ft	9	167 m 548 ft Standing
1908	Singer Building			187 m	612 ft	47	Demolished in 1968
1909	Met Life Tower			213 m	700 ft	50	Standing
1913	Woolworth Building	New York City		241 m	792 ft	57	Standing
1930	40 Wall Street				70	283 m 927 ft	Standing
1930	Chrysler Building			282.9 m	927 ft	77	319 m 1,046 ft Standing
1931	Empire State Building			381 m	1,250 ft	102	443 m 1,454 ft Standing
1972	World Trade Center (North Tower)			417 m	1,368 ft	110	527.3 m 1,730 ft Destroyed in 2001 in the September 11 attacks
1974	Willis Tower (formerly Sears Tower)	Chicago		442 m	1,450 ft	108	527 m 1,729 ft Standing
1996	Petronas Towers	Kuala Lumpur	 Malaysia	379 m	1,242 ft	88	452 m 1,483 ft Standing
2004	Taipei 101	Taipei	 Taiwan	449 m	1,474 ft	101	509 m 1,671 ft Standing
2010	Burj Khalifa	Dubai	 United Arab Emirates	828 m	2,717 ft	163	829.8 m 2,722 ft Standing

DATES	POLLSTER	GRADE	SAMPLE	WEIGHT	APPROVE / DISAPPROVE		ADJUSTED
					APPROVE	DISAPPROVE	
• DEC. 28-30	Gallup	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B-</span>	1,500 A	 1.03	40%	55%	41% 53%
• DEC. 26-28	Rasmussen Reports/Pulse Opinion Research	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">C+</span>	1,500 LV	 0.85	45%	53%	40% 53%
• DEC. 24-28	Ipsos	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">A-</span>	1,519 A	 2.01	37%	58%	37% 57%
• DEC. 23-27	Gallup	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B-</span>	1,500 A	 0.58	38%	56%	39% 54%
• DEC. 24-26	YouGov	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B</span>	1,500 A	 1.13	38%	52%	39% 55%

# Scraping HTML tables

- HTML tables are everywhere.
- They are easy to spot in the wild - just look for `<table>` tags!
- Exactly because scraping tables is an easy and repetitive task, there is a dedicated `rvest` function for it: `html_table()`.

## Function definition

```
R> html_table(x,  
+   header = NA,  
+   trim = TRUE,  
+   dec = ".,"  
+   na.strings = "NA",  
+   convert = TRUE  
+ )
```

Argument	Description
<code>x</code>	Document (from <code>read_html()</code> ) or node set (from <code>html_elements()</code> ).
<code>header</code>	Use first row as header? If <code>NA</code> , will use first row if it consists of <code>&lt;th&gt;</code> tags.
<code>trim</code>	Remove leading and trailing whitespace within each cell?
<code>dec</code>	The character used as decimal place marker.
<code>na.strings</code>	Character vector of values that will be converted to <code>NA</code> if <code>convert</code> is <code>TRUE</code> .
<code>convert</code>	If <code>TRUE</code> , will run <code>type.convert()</code> to interpret texts as int, dbl, or <code>NA</code> .

# Scraping HTML tables: example

- We are going to scrape a small table from the Wikipedia page [https://en.wikipedia.org/wiki/List\\_of\\_human\\_spaceflights](https://en.wikipedia.org/wiki/List_of_human_spaceflights).
- (Note that we're actually using an old version of the page (dating back to May 1, 2018), which is accessible [here](#).  
Wikipedia pages change, but this old revision and associated link won't.)
- The table is not entirely clean: There are some empty cells, but also images and links.
- The HTML code looks straightforward though.

The screenshot shows the Wikipedia page 'List of human spaceflights'. At the top, there's a sidebar with various links like 'Contents', 'Featured content', 'Current events', 'Recent changes', 'Donate to Wikipedia', and 'Help'. Below the sidebar is the main content area. A summary table at the top right shows the number of flights for different countries over time. To the right of the table is a bar chart titled 'Orbit of all humans launched into space as of December the 31st, 2016'. Below the chart is a photograph of the Apollo 7 mission. The bottom of the page features a footer with links to print/export options and language versions.

Period	Russia USSR	United States	China	Total
1961–1970	16	25	41	
1971–1980	30	8	38	
1981–1990	25	38	63	
1991–2000	20	63	83	
2001–2010	24	34	3	61
2011–2020	24	3	3	30
Total	139	171	6	316

This screenshot shows the same Wikipedia page with the developer tools (Elements) open, focusing on the table structure. The table is identified by the class 'wikitable' and has a style attribute of 'text-align: right;'. The table body contains several rows of data, each with four columns representing Russia/USSR, United States, China, and Total. The developer tools also show the CSS styles applied to the table, including margin, border, and padding values.

Period	Russia USSR	United States	China	Total
1961–1970	16	25	41	
1971–1980	30	8	38	
1981–1990	25	38	63	
1991–2000	20	63	83	
2001–2010	24	34	3	61
2011–2020	24	3	3	30
Total	139	171	6	316

```
<table class="wikitable" style="text-align:right;">
  <tbody>
    <tr>...</tr>
    <tr>...</tr>
    <tr>
      <td>...</td>
      <td>30</td>
      <td>8</td>
      <td>38</td>
    </tr>
    <tr>...</tr>
    <tr>...</tr>
    <tr>...</tr>
    <tr>...</tr>
    <tr>...</tr>
  </tbody>
</table>
```

# Scraping HTML tables: example (cont.)

```
R> library(rvest)
R> url <- "https://en.wikipedia.org/wiki/List_of_human_spaceflights"
R> url_p <- read_html(url)
R> tables <- html_table(url_p, header = TRUE)
R> spaceflights <- tables[[1]]
R> spaceflights

## # A tibble: 7 × 5
##   ``        `Russia &nbsp; Soviet Union` `United States` China Total
##   <chr>    <chr>                <chr>      <int> <chr>
## 1 1961–1970 16                  25          NA 41
## 2 1971–1980 30                  8           NA 38
## 3 1981–1990 *25                *38         NA *63
## 4 1991–2000 20                  63          NA 83
## 5 2001–2010 24                  34          3  61
## 6 2011–2020 24                  3           3  30
## 7 Total     *139                *171         6 *316
```

# Web scraping: good practice

---

# Scraping: the rules of the game

1. You take all the responsibility for your web scraping work.
2. Think about the nature of the data. Does it entail sensitive information? Do not collect personal data without explicit permission.
3. Take all copyrights of a country's jurisdiction into account. If you publish data, do not commit copyright fraud.
4. If possible, stay identifiable. Stay polite. Stay friendly. Obey the scraping etiquette.
5. If in doubt, ask the author/creator/provider of data for permission—if your interest is entirely scientific, chances aren't bad that you get data.

# Consult robots.txt

## What's robots.txt?

- "Robots exclusion standard", informal protocol to prohibit web robots from crawling content
- Located in the root directory of a website (e.g., [google.com/robots.txt](http://google.com/robots.txt))
- Documents which bot is allowed to crawl which resources (and which not)
- Not a technical barrier, but a sign that asks for compliance

## What's robots.txt?

- Not an official W3C standard
- Rules listed bot by bot
- General rule listed under `User-agent: *` (most interesting entry for R-based crawlers)
- Directories/folders listed separately

### Example

```
User-agent: Googlebot  
Disallow: /pics/  
Disallow: /private/
```

### Universal ban

```
User-agent: *  
Disallow: /
```

### Allow declaration

```
User-agent: *  
Disallow: /pics/  
Allow: /pics/public/
```

### Crawl delay (in seconds)

```
User-agent: *
```

# Downloading HTML files

## Stay modest when accessing lots of data

- Content on the web is publicly available.
- But accessing the data causes server traffic.
- Stay polite by querying resources as sparsely as possible.

## Two easy-to-implement practices

1. Do not bombard the server with requests - and if you have to, do so at modest pace.
2. Store web data on your local drive first, then parse.

## Looping over a list of URLs

```
R> for (i in 1:length(list_of_urls)) {  
+   if (!file.exists(paste0(folder, file_names[i])))  
+     download.file(list_of_urls[i],  
+                   destfile = paste0(folder, file_n.  
+                                         )  
+     Sys.sleep(runif(1, 1, 2))  
+   }  
+ }
```

- `!file.exists()` checks whether a file does not exist in the specified location.
- `download.file()` downloads the file to a folder. The destination file (location + name) has to be specified.
- `Sys.sleep()` suspends the execution of R code for a given time interval (in seconds).

# Staying identifiable

## Don't be a phantom

- Downloading massive amounts of data may arouse attention from server administrators.
- Assuming that you've got nothing to hide, you should stay identifiable beyond your IP address.

## Two easy-to-implement practices

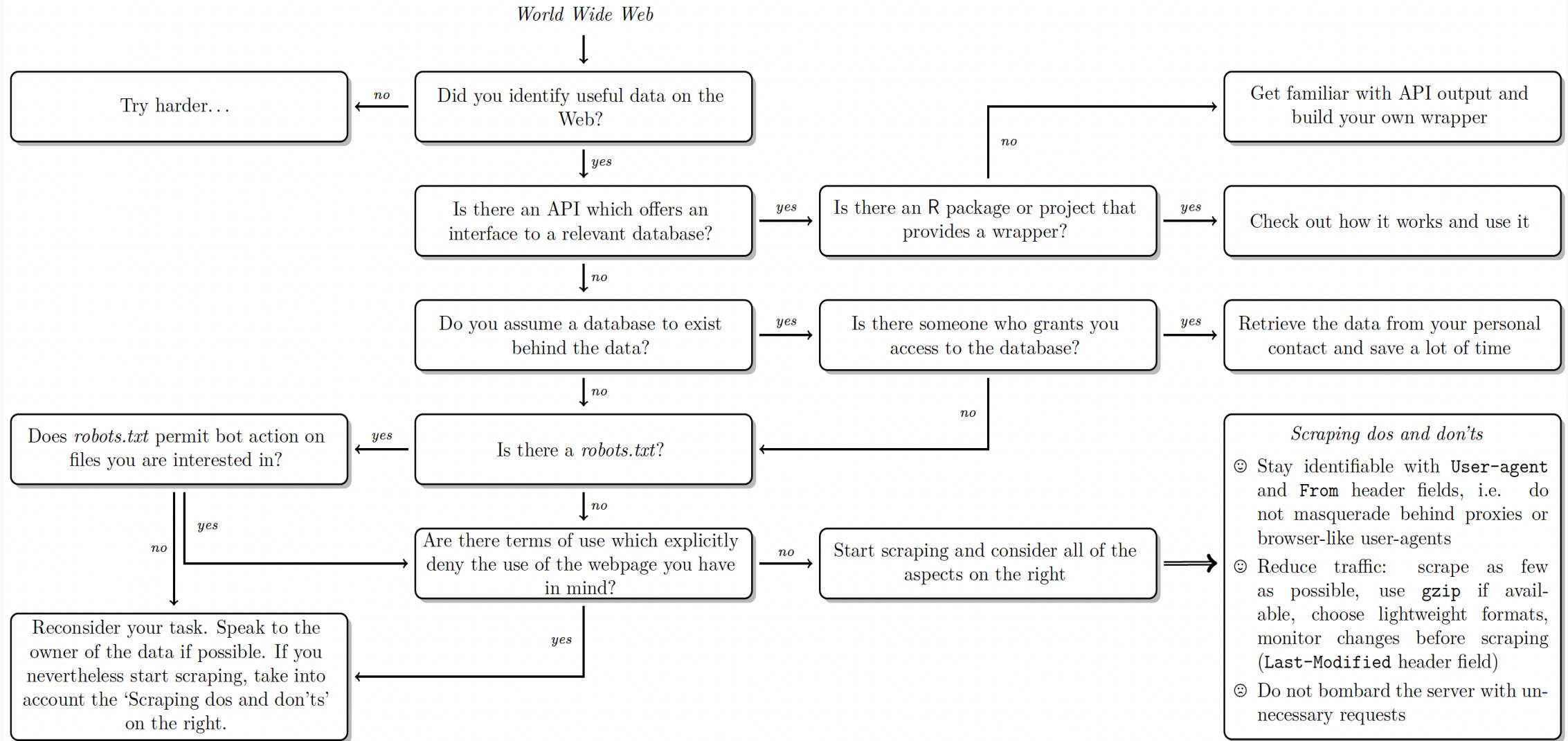
1. Get in touch with website administrators / data owners.
2. Use HTTP header fields `From` and `User-Agent` to provide information about yourself (by passing these to `add_headers()` from the `httr` library).

## Staying identifiable in practice

```
R> url <- "http://a-totally-random-website.com"
R> rvest_session <- session(url,
+   add_headers(From = "my@email.com",
+               `UserAgent` =
+                 R.Version()$version.string
+             )
+ )
R> headlines <- rvest_session %>%
+   html_elements(xpath = "p//a") %>%
+   html_text()
```

- `rvest`'s `session()` creates a session object that responds to HTTP and HTML methods.
- Here, we provide our email address and the current R version as `User-Agent` information.
- This will pop up in the server logs: The webpage administrator has the chance to easily get in touch with you.

# Scraping etiquette (cont.)



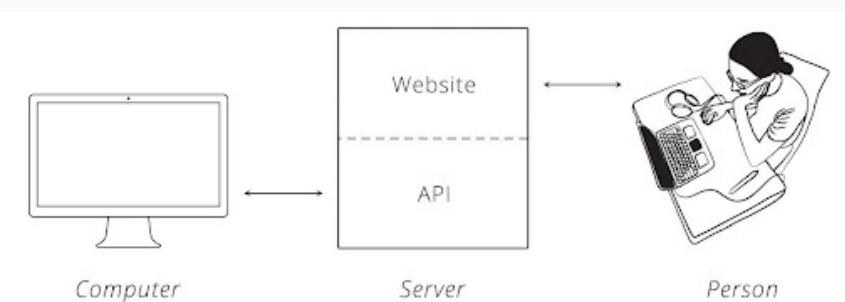
# Web APIs: the basics

---

# What are web APIs?

## Definition

- A web **A**pplication **P**rogramming **I**nterface lets you/your program query a provider for specific data.
- Think of web APIs as "data search engines": You pose a request, the API answers with a bulk of data.
- Many popular web services provide APIs (Google, Twitter, Wikipedia, ...).
- Often, APIs provide data in JSON, XML (can be any format though).



credit Brian Cooksey

## Key concepts

- **Server:** A computer that runs an API and can be talked to.
- **Client:** A program that exchanges data with a server through an API.
- **Protocol:** The rule set underlying how computers talk to each other (e.g. HTTP).
- **Method:** The "verbs" that clients use to talk with a server (in HTTP speak: `GET`, `POST`, and others).
- **Endpoint:** URLs that can be specified in a particular way to query the API database.
- **Request:** What the client asks of the server (see Methods above).
- **Response:** The server's response. This includes a *Status Code* (e.g. "200", "404"), a *Header* (meta-information about the response), and a *Body* (the actual content that we're interested in).

# What do APIs do?

## Key perks from a web scraping perspective

- APIs provide instance access to clean data.
- They free us from building manual scrapers.
- They make it easier for a computer to interact with data on server.
- API usage implies mutual agreement about data collection.



## Restaurant analogy<sup>1</sup>

Restaurant	Web API
You, the client	Your program/computer
The restaurant	API provider
The waiter	API
The menu	API documentation
Your order	Specified API endpoint (request)
The kitchen	API database
The food	API response
The bill	API pricing

<sup>1</sup> Kitchen example inspired by Jason Johl

# Why do organizations have APIs?

## Scalability and regulation of access

- Imagine if everyone decided to retrieve data from a server in an unstructured way.  
The amount of energy that this would consume would be very high.
- Also, access to data would be largely unregulated.
- With APIs, organizations can provide a regulated and organized way for clients to retrieve a large amount of data, without overwhelming or crashing their server, or violating their Terms of Use.



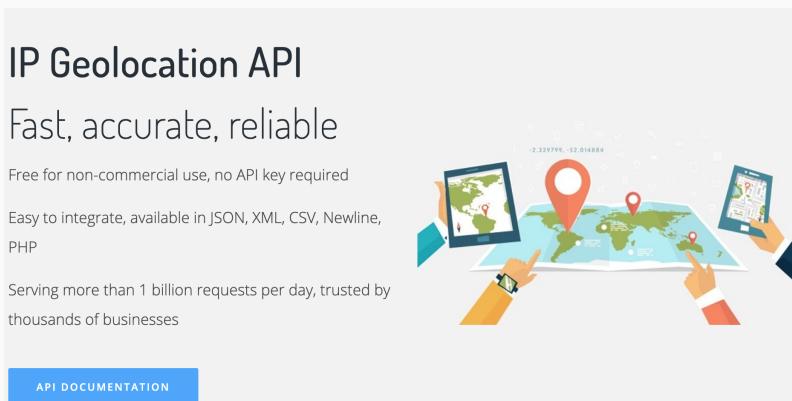
## More reasons

- **Monetization:** Data can be turned into a sellable product.
- **Innovation:** Clients have access to data and develop their own products and solutions.
- **Expansion:** APIs can help companies move to different markets or partner with other companies.

# Example: Working with the IP API

## The IP API

- The IP API at <https://ip-api.com/> takes IP addresses and provides geolocation data (latitude, longitude, but also country state, city, etc.) in return.
- This is useful if you want to, e.g., map IP address data (although this is **not perfectly accurate**).
- The API is free to use and requires no registration. (There's also a pro service for commercial use though.)



## An example JSON response

API Demo

Search any IP address/domain

95.90.241.209

SEARCH

```
{  
  "query": "95.90.241.209",  
  "status": "success",  
  "continent": "Europe",  
  "continentCode": "EU",  
  "country": "Germany",  
  "countryCode": "DE",  
  "region": "BE",  
  "regionName": "Land Berlin",  
  "city": "Berlin",  
  "district": "",  
  "zip": "10117",  
  "lat": 52.5203,  
  "lon": 13.3849,  
  "timezone": "Europe/Berlin",  
  "offset": 3600,  
  "currency": "EUR",  
  "isp": "Vodafone Kabel Deutschland",  
  "org": "Vodafone Kabel Deutschland GmbH",  
  "as": "AS3209 Vodafone GmbH",  
  "asname": "VODANET",  
  "mobile": false,  
  "proxy": false,  
  "hosting": false  
}
```

A detailed map of Berlin, Germany, showing various neighborhoods and landmarks. A red marker indicates the location corresponding to the IP address 95.90.241.209, which is located in the central part of the city.

# Example: Documentation

## Overview

### Overview

This documentation is intended for developers who want to write applications that can query IP-API. We serve our data in multiple formats via a simple URL-based interface over HTTP, which enables you to use our data directly from a user's browser or from your server.

### Geolocation API

Response formats

[JSON](#)

[XML](#)

[CSV](#)

[Newline](#)

[PHP](#)

### Batch API

Query multiple IP addresses in one HTTP request.

[Batch JSON](#)

## JSON endpoint

The API base path is

`http://ip-api.com/json/{query}`

`{query}` can be a single IPv4/IPv6 address or a domain name. If you don't supply a query the current IP address will be used.

### Parameters

Query parameters (such as custom fields and JSONP callback) are appended as GET request parameters, for example:

`http://ip-api.com/json/?fields=61439`

<code>fields</code>	response fields <i>optional</i>
<code>lang</code>	response language <i>optional</i>
<code>callback</code>	wrap inside (JSONP) <i>optional</i>

There is no API key required.

### Quick test

You can edit this query and experiment with the options

GET

`http://ip-api.com/json/24.48.0.1`

SEND

# Example: Calling the API with R

First, we specify the endpoint. We use the JSON endpoint and start with an empty query field.

```
R> endpoint <- "http://ip-api.com/json"
```

Next, we call the API with `httr`'s `GET()` method.

```
R> endpoint <- "http://ip-api.com/json"
R> response <- httr::GET(endpoint)
R> response

## Response [http://ip-api.com/json]
##   Date: 2022-10-30 20:45
##   Status: 200
##   Content-Type: application/json; charset=utf-8
##   Size: 312 B
```

Hooray, we successfully called the API and got JSON data in return! We inspect the content with `httr`'s `content()` function.

```
R> response_parsed <- httr::content(response)
R> response_parsed
```

```
## $status
## [1] "success"
##
## $country
## [1] "Germany"
##
## $countryCode
## [1] "DE"
##
## $region
## [1] "BE"
##
## $regionName
## [1] "Land Berlin"
##
```

# Example: Calling the API with R *cont.*

This looks like an R list, which is useful. `httr::content()` automatically parsed the JSON file into an R list, which is useful. We could have also kept the raw (here: text) content with `httr::content(as = "text")`.

For more convenience (and flexibility), we can also use the powerful `jsonlite` package and its parser:

```
R> response_parsed <- jsonlite::fromJSON(endpoint)
R> response_parsed

## $status
## [1] "success"
##
## $country
## [1] "Germany"
##
## $countryCode
## [1] "DE"
##
## $region
```

Given that the data structure is low-dimensional, we can easily map it onto a data frame:

```
R> as.data.frame(response_parsed)

##   status country countryCode
## 1 success Germany        DE
##   region regionName    city
## 1      BE Land Berlin Berlin
##     zip      lat      lon
## 1 10117 52.5203 13.3849
##           timezone
## 1 Europe/Berlin
##                   isp
## 1 Vodafone Kabel Deutschland
##                   org
## 1 Vodafone Kabel Deutschland GmbH
##                   as
## 1 AS3209 Vodafone GmbH
##           query
## 1 95.90.241.209
```

# Example: Calling the API with R *cont.*

We can easily modify the call to retrieve more data:

```
R> endpoint ← "http://ip-api.com/json/91.198.174.1"
R> response_parsed ← jsonlite::fromJSON(endpoint)
R> response_parsed

## $status
## [1] "success"
##
## $country
## [1] "Netherlands"
##
## $countryCode
## [1] "NL"
##
## $region
## [1] "NH"
##
## $regionName
## [1] "North Holland"
##
## $city
```

The API also allows for batch processing. This gives you the ability to query multiple IP addresses in one HTTP request, which is significantly faster than submitting individual queries. To that end, the batch of IP addresses or domains must be sent using a `POST` request:

```
R> endpoint ← "http://ip-api.com/batch"
R> response ← httr::POST(url = endpoint, body =
+   '[{"208.80.152.201", "91.198.174.192"}]', encode
R> httr::content(response, as = "text") %>%
+   jsonlite::fromJSON() %>%
+   as.data.frame()

## #> #>   status      country countryCode region      regionName
## #> #>   1 success United States          US      IL      Illinois
## #> #>   2 success Netherlands          NL      NH North Holland
## #> #>   #>   lat        lon      timezone
## #> #>   #>   1 41.8781 -87.62980 America/Chicago Wikimedia Foundation
## #> #>   #>   2 52.3676  4.90414 Europe/Amsterdam Wikimedia Europe
## #> #>   #>   #>   org
## #> #>   #>   1 Wikimedia Foundation Inc AS14907 Wikimedia Foundation
```

# Accessing APIs with R

## Is there a pre-built API client for R?

1. **Yes.** Great. Use it if it provides the functionality you need (if not, see option 2).
2. **No.** Build your own API client.



## API clients

- Provide an interface to APIs
- Hide the API back-end
- Let you stay in your programming environment
- Good news: In most cases, this will be the world you live in
- Please always cite package authors when you use their work. Run `citation("<package name>")` to see how.

## Building a client from scratch

- Dive into the API documentation
- Write your own functions to specify endpoint calls and turn API output into R data structures
- Build infrastructure for API authentication if necessary

# API clients: example

The `ipapi` package by Bob Rudis provides high-level access to the IP API. You just have to install it and can then work with the `geolocate()` function provided by the package to call the API.

```
R> # devtools::install_github("hrbrmstr/ipapi") # uncomment to install if necessary
R> library(ipapi)
R> ip_df <- geolocate(c("", "10.0.1.1", "72.33.67.89", "www.spiegel.de"), .progress=TRUE)
R> ip_df
```

```
##      status      country countryCode region   regionName           city    zip
## 1: success     Germany          DE     BE Land Berlin            Berlin 10117
## 2: fail        <NA>          <NA>  <NA>       <NA>            <NA>  <NA>
## 3: success United States        US     WI  Wisconsin         Madison 53706
## 4: success     Germany          DE     HE      Hesse Frankfurt am Main 60314
##      lat      lon      timezone           isp
## 1: 52.5203 13.3849 Europe/Berlin Vodafone Kabel Deutschland
## 2:     NA      NA          <NA>            <NA>
## 3: 43.0713 -89.4063 America/Chicago University of Wisconsin Madison
## 4: 50.1103  8.7147 Europe/Berlin           Link11 GmbH
##      org           as
## 1: Vodafone Kabel Deutschland GmbH AS3209 Vodafone GmbH
## 2:                      <NA>            <NA>
## 3: University of Wisconsin Madison AS59 University of Wisconsin Madison
```

# Restricted API access

## Why API access can be restricted

- The service provider wants to know who uses their API.
- Hosting APIs is costly. API usage limits can help control costs.
- The API hoster has a commercial interest: You pay for access (sometimes for advanced features or massive queries only).

## Access tokens

- Access tokens serve as keys to an API.
- They usually come in form of a randomly generated string, such as `dk5nSj485jJZP3847kjU`.
- Obtaining a token requires registration; sometimes payment. Sometimes disclosure of your intentions.
- Once you have the token, you pass it along with your regular API call using a key parameter.

# Restricted API access: example

The *New York Times* provides several APIs for developers at <https://developer.nytimes.com/>. In order to use them, we have to register as a developer (for free) and register our app. Then, we can use that key to call one of the APIs.

**{T} Developers**

Home APIs Covid-19 Data Get Started [REDACTED]

**APIs**

 <b>Archive API</b> Get all NYT article metadata for a given month.	 <b>Article Search API</b> Search for New York Times articles.	 <b>Books API</b> Get NYT Best Sellers Lists and lookup book reviews.	 <b>Most Popular API</b> Popular articles on NYTimes.com.
 <b>Movie Reviews API</b> Search for movie reviews.	 <b>RSS Feeds</b> NYT RSS section feeds.	 <b>Semantic API</b> Get semantic terms (people, places, organizations, and locations).	 <b>Times Tags API</b> NYT controlled vocabulary.

**{T} Developers**

Home APIs Covid-19 Data Get Started [REDACTED]

**RDataCollection**

**Overview**

App Name *	RDataCollection
Description	Package to do article search
App ID	[REDACTED]

**API Keys**

Key	Secret	Status	Created	Expires	Actions
[REDACTED]	[REDACTED]	Show secret	✓ Active	Feb 1, 2019, 12:18 PM	never Revoke

**ADD KEY**

**APIs \***

Name	Description	Status	Actions
Archive API	Get all NYT article metadata for a given month.	—	Enable
Article Search API	Search for New York Times articles.	✓ Enabled	Disable
Books API	Get NYT Best Sellers Lists and lookup book reviews.	—	Enable
Most Popular API	Popular articles on NYTimes.com.	✓ Enabled	Disable
Movie Reviews API	Search for movie reviews.	—	Enable
RSS Feeds	NYT RSS section feeds.	—	Enable

# Restricted API access: example *cont.*

First, we load the key that we stored separately as a string (here: `nytimes_apikey` stored in `rkeys.RDa`).

```
R> load("/Users/simonmunzert/rkeys.RDa")
```

Next we specify the API endpoint using the API key:

```
R> endpoint ← "https://api.nytimes.com/svc/mostpopular/v2/viewed/1.json?"  
R> url ← paste0(endpoint, "api-key=", nytimes_apikey)
```

Finally, we call the API and inspect the results:

```
R> nytimes_most_popular ← jsonlite::fromJSON(url)  
R> nytimes_most_popular$results$title[1:3]  
  
## [1] "Elon Musk Is Said to Have Ordered Job Cuts Across Twitter"  
## [2] "Jerry Lee Lewis, a Rock 'n' Roll Original, Dies at 87"  
## [3] "Tom Brady and Gisele Bündchen Divorce After 13 Years"
```

# Restricted API access: some advice

**NEVER** hard-code your personal API key (or any personal information for that matter) in R scripts that you plant to stare.

Instead, use one of the following options:

1. Store your API keys in a separate file that you store somewhere else and only import for the purpose of using them (see previous example).
2. Store your API keys in environment variables.

For the second option, you can use `sys.setenv()` and `sys.getenv()` as in:

```
R> ## Set new environment variable called MY_API_KEY. Current session only. Don't store code in script.  
R> Sys.setenv(MY_API_KEY="abcdefghijklmnopqrstuvwxyz0123456789")  
R>  
R> ## Assign the environment variable to an R object and pass it on where needed.  
R> my_api_key = Sys.getenv("MY_API_KEY")
```

The downside of this approach is that this environment variable will not persist across sessions. To make it persistent (on your system), you should modify the `.Renviron` file. Check out [these instructions](#) to learn how that works.

# Recap: tapping APIs with R

1. Figure out whether an API is available that serves your needs.
2. Figure out whether an up-to-date and fully functional R client for API is available. Make yourself familiar with the API Terms of Use and client functionality, then use it.
3. If no client is available, build your own.
  - a. Dive into the documentation.
  - b. Use `httr` package to construct requests.
  - c. Use `jsonlite` package to parse the JSON response (or other parsers such as `xml2`, depending on API output).
  - d. Address API error handling, user agent, authentication, pagination issues. (See [here](#) for more info.)
  - e. Write useful high-level functions that wrap your API calls and parsing operations.
  - f. Consider publishing your R API client as a package.

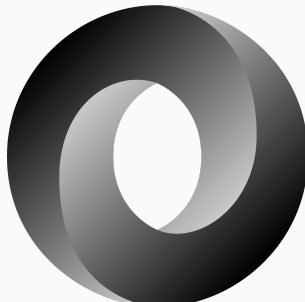
# JSON

---

# What's JSON?

## Quick facts

- **JavaScript Object Notation**
- Popular data exchange format for web services / APIs
- "the fat-free alternative to XML"
- JSON ≠ Java, but a subset of JavaScript
- However, very flexible and not dependent upon any programming language
- Import of JSON data into R is relatively straightforward with the `jsonlite` package



# Example

```
[  
  {  
    "name" : "van Pelt, Lucy",  
    "sex" : "female",  
    "age" : 32  
  },  
  {  
    "name" : "Peppermint, Patty",  
    "sex" : "female",  
    "age" : null  
  },  
  {  
    "name" : "Brown, Charlie",  
    "sex" : "male",  
    "age" : 27  
  }  
]
```

# Basic syntax

## Types of brackets

1. Curly brackets, `{` and `}`, embrace objects. Objects work similar to elements in XML/HTML and can contain other objects, key-value pairs or arrays.
2. Square brackets, `[` and `]`, embrace arrays. An array is an ordered sequence of objects or values.

## Data structure

JSON can map complex data structures (objects nested within objects etc.), which can make it difficult to convert JSON into flattened R data structures (e.g., a data frame).

Also, there is no ultimate JSON-to-R converting function.

Luckily, JSON files returned by web services are usually not very complex. And `jsonlite` simplifies matters a lot.

## Key-value pairs

Keys are put in quotation marks; values only if they contain string data.

```
"name" : "van Pelt, Lucy"  
"age" : 32
```

Keys and values are separated by a colon.

```
"age" : 32
```

Key-value pairs are separated by commas.

```
{"name" : "van Pelt, Lucy", "age" : 32}
```

Values within arrays are separated by commas.

```
["van Pelt, Lucy", "Peppermint, Patty"]
```

# JSON and R

## Parsing JSON with R

- There are different packages available for JSON parsing with R.
- Choose `jsonlite` by Jeroen Ooms: It's well maintained and provides convincing mapping rules.

## Key functions

There are two key functions in `jsonlite`:

- `fromJSON()`: converts input from JSON data into R objects following a set of **conventions**.
- `toJSON()`: converts input from R objects into JSON data

Get started with the package following [this vignette](#).

## Conversion rules of `jsonlite`

```
R> library(jsonlite)
R> x <- '[1, 2, true, false]'
R> fromJSON(x)
```

```
## [1] 1 2 1 0
```

```
R> x <- '["foo", true, false]'
R> fromJSON(x)
```

```
## [1] "foo"    "TRUE"   "FALSE"
```

```
R> x <- '[1, "foo", null, false]'
R> fromJSON(x)
```

```
## [1] "1"      "foo"    NA       "FALSE"
```

# Summary

---

# More on web scraping

Until now, the toy scraping examples were limited to single HTML pages. However, often we want to **scrape data from multiple pages**. You might think of newspaper articles, Wikipedia pages, shopping items and the like. In such scenarios, automating the scraping process becomes really powerful. Also, principles of polite scraping are more relevant then.

In other cases, you might be confronted with

- forms,
- authentication,
- dynamic (JavaScript-enriched) content, or want to
- automatically navigate through pages interactively.

There's only so much we can cover in one session. Check out more material online [here](#) and [there](#) to learn about solutions to some of these problems.

# More on web APIs

Collecting data from the web using APIs provided by the data owner represents the **gold standard of web data retrieval**. It allows for pure data collection without "HTML layout waste", standardized data access, de facto agreement on data collection by the data owner, and robustness and scalability of data collection.

On the other hand, the rise of API architectures is not without issues. It requires knowledge of the architecture and creates dependencies on API suppliers. While APIs have the potential to make data access more democratic, they can also add to the siloing of information.

If you want to learn more about APIs in depth, check out [this introduction to APIs](#). Or, check out this [hands-on intro video to APIs in R by Leo Glowacki](#).

There are many resources that give an overview of existing public APIs, including [ProgrammableWeb](#) and this [List of public APIs on GitHub](#). Another useful resource is [APIs for social scientists - a collaborative review](#).

Finally, if you plan to write an R client for a web API, check out [this guide](#).

# Coming up

## Assignment

Assignment 3 is about to go online. Check it out and start scraping the web (politely).

## Next lecture

Model fitting and evaluation. Now that we know how to retrieve data, let's learn how to model and learn from them.