How to Collect Web Data via APIs

FELIX SOLDNER, N. GIZEM BACAKSIZLAR TURBIC, POURIA MIRELMI, JULIAN KOHNE

GESIS – Leibniz Institute for the Social Sciences, Cologne, Germany

Publication date: August 27, 2024; Version 1.0

In this guide, we discuss and show how to collect web data by using APIs (Application Programming Interfaces). We will use Wikipedia as an example and show how we can obtain articles and extract information from this platform using an API. This guide is written for users who want to collect or learn about APIs and have a basic understanding of Python programming.

Keywords: automated data collection, API, Wikipedia, Python

1 What are APIs?

An Application Programming Interface (API) is “a way for two or more computer programs or components to communicate with each other. It is a type of software interface, offering a service to other pieces of software” (Reddy, 2011). An API supports communication between computers by providing pre-written code that can be reused to streamline the interaction. For example, code packages in the programming language Python, such as pandas are APIs because they assist programmers in speeding up their coding by providing a set of pre-programmed functions that perform commonly needed operations. Similarly, typing a domain name in a browser uses an API and supports the retrieval of information from a server (accessing a website) without the need to provide an IP address. APIs that operate across the web are called **web APIs** and can be used through a browser (Figure 1). However, manually making API calls with a browser by constructing a

URL, which functions as a command to retrieve information from a server, would be time-consuming for large-scale data collection. Alternatively, we can use so-called "wrappers" that act as an overlay to the API and construct the commands for us in a programming language, thus, streamlining API calls.

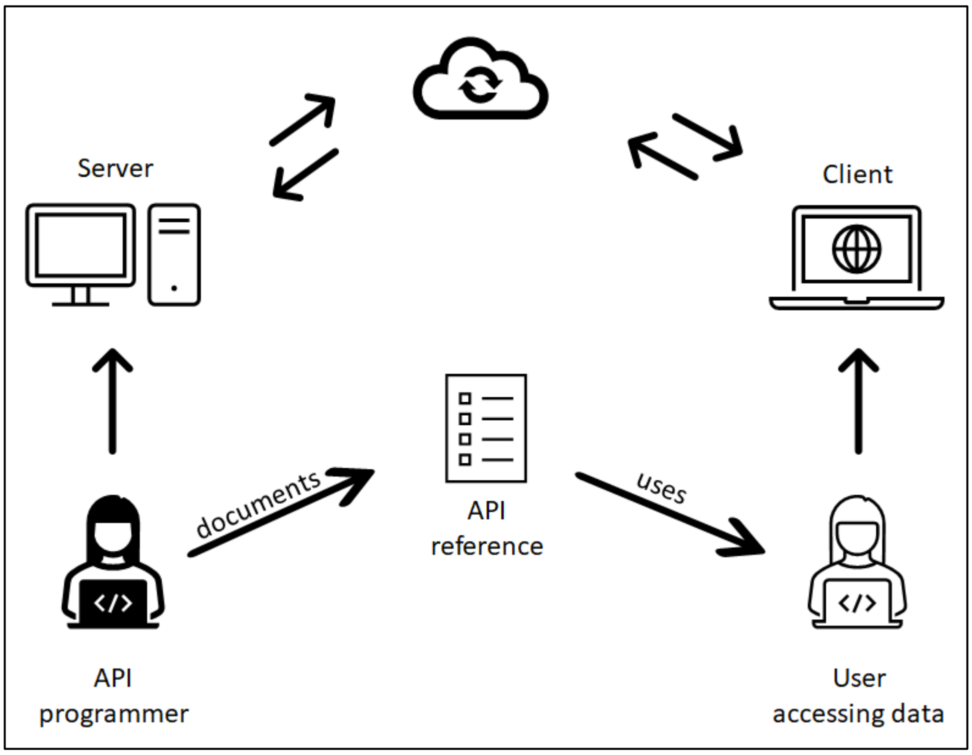

Figure 1. Relational setup of using an API

Web platforms provide APIs to users to make their interaction with the website more accessible by allowing automated uploads, downloads, and content changes. For example, eBay allows users to access their shops' inventory automatically, or YouTube allows users to upload and change content as well as access video reactions (e.g., views, comments). In social science research contexts, we are interested in APIs, which also provide platform data, often related to social interactions on social media websites. In many cases, researchers who want to collect data via platform-specific APIs must create a developer account for the platform, apply for API access, and often also need to undergo a vetting procedure in which the goals and procedures of the project must be described.

Most modern social media platforms offer APIs (Perriam et al., 2020), but not all platforms have an API or provide public access. In addition, API accesses are often tiered into free options that provide less information (e.g., type and scope) and more costly access levels that provide more data. Very often, restrictions, typically in the form of so-called rate limits, are introduced or altered over time. For example, Reddit recently limited the number of API requests per minute (Huffman, 2023), YouTube has increasingly limited its daily API quotas (Google for Developers, 2022), and Instagram and Facebook have also heavily restricted the accessibility of their APIs in the past (Abouelhassan, 2024). Hence, while APIs often enable researchers to easily access public data from social media platforms, they rarely provide access to all their data in unrestricted quantities – Wikipedia being a notable exception in that regard. Accordingly, some information may be visible to researchers in a browser but not available through the corresponding API.

Often, but not always, APIs and the associated wrappers are documented, describing their functions, commands, and what type of information can be accessed and retrieved. The documentation is an important source of information and should be consulted whenever possible. Additionally, the Terms of Use/Service (ToS) of the API and the platform should be carefully read as they lay out how the data can be used (and whether or under what conditions they may be shared or published).

2 Using APIs vs. using web scraping

An alternative to using APIs is web scraping, sometimes also called screen scraping. In essence, this procedure means that instead of directly querying a platforms database through an API, researchers access the website containing the desired information through a browser and save (parts of) the HTML file used by the browser. In comparison to using an API, we can identify advantages but also disadvantages in web scraping.

The main advantage of web scraping is, that in principle, all data that is visible in the browser can also be collected by researchers. This includes data that might not be available through the API, information on how things are displayed to users, or content that is only visible after logging in with an account. Unless explicitly prevented by platforms through technical means, researchers can also access as much data as their browser session can handle at a time.

The main 'disadvantage' of web scraping is that many platforms' ToS do not permit it. Whether this restriction is legally binding for research purposes is being debated – and can depend on a variety of factors, such as the volume of data scraped, what the data is used for, or how it is being archived (Klawonn, 2020).

In essence, our advice would be to use an API whenever possible and only use web scraping if the required data cannot be accessed otherwise. Soldner (2024) offers a detailed overview on the advantages and disadvantages of APIs and web scraping and the upcoming Guides #10 and #11 will provide practical guidance on how to implement web scrapers for static and dynamic webpages (i.e., whether a webpage loads content after user interactions, such as scrolling).

3 How to work with the Wikimedia (Wikipedia) API

As an example, and a use case for this guide, we are interested in obtaining information from Wikipedia, the free online encyclopedia, created and edited by volunteers and one of the products of the Wikimedia Foundation. Wikimedia offers a free API that can be used without requiring a developer account. The API can be used through a web browser and via the documentation. A sandbox environment can be accessed, where the URL creation for API calls, which allow accessing Wikipedia content (articles, revisions, discussions, etc.), can be tested. For content creation on Wikipedia, such as editing, an account is required.

Since working with browser URLs is tedious, we will make use of API wrappers, such as the Python package Wikipedia. Before using this wrapper, we need to install the package through conda or pip, depending on the programming environment. Information on how to install packages with conda can be found here and with pip here. For the Wikipedia package we use the command `pip install wikipedia` to install the package.

1.1 Retrieving basic information

We start by importing the Wikipedia wrapper.

```
import wikipedia as wp
```

Example: A search query with `wikipedia` can be made with `search()`. We want to search for the city "Seattle".

```
wp.search("seattle")
```

```
['Seattle',
 'Seattle Seahawks',
 'Seattle Sounders FC',
 'Seattle Kraken',
 'Seattle Mariners',
 'Chief Seattle',
 'The Seattle Times',
 'Seattle metropolitan area',
 'Sleepless in Seattle',
 'Seattle Post-Intelligencer']
```

We receive a list of found articles (also called pages) about the city and other related subjects. We can reduce the results by specifying `results` in the function.

```
wp.search("seattle", results=3)
```

```
['Seattle', 'Seattle Sounders FC', 'Seattle Seahawks']
```

Since we are interested in Seattle, the city, we use the first listed page, `'Seattle'`. We can retrieve a summary of the article through `summary()`.

```
wp.summary("Seattle")
```

```
'Seattle (see-AT-əl) is a seaport city on the West Coast of the
United States. It is the seat of King County, Washington. With
a 2022 population of 749,256 it is the most populous city in
both the state of Washington and the Pacific Northwest region
of North America, and […]'
```

We can limit the number of retrieved sentences via the variable `sentences`.

```
wp.summary("Seattle", sentences=1)
```

`summary()` will raise a `DisambiguationError` if the page is a disambiguation page or a `PageError` if the page doesn't exist. However, by default, `summary()` will try to find the page we want and provide a summary of a suggested page (similar spelling, etc.).

For example, we can try accessing a summary for "Mercury".

```
wp.summary("Mercury")
```

The function gives us a `DisambiguationError`. We can work around it through `try` and `except` and obtain the possible page options through `options`.

```
try:
    wp.summary("Mercury")
except wp.exceptions.DisambiguationError as DisError:
    article_options = DisError.options
print(article_options)
```

```
['Mercury (planet)', 'Mercury (element)', 'Mercury
(mythology)', 'Mercury (company)', […] ]
```

We can now access the page we are interested in and retrieve its summary. Let us pick the element (second in the list).

```
wp.summary(article_options[1])
```

```
"Mercury is a chemical element; it has symbol Hg and atomic
[…]"
```

Let us explore how we can retrieve information from the pages as we see them on the web. As a reference, have a look at the Wikipedia page about Seattle: https://en.wikipedia.org/wiki/Seattle.

First, we can use the function `WikipediaPage()` to load and access data from full Wikipedia pages. We start by giving the function the page name we are interested in. We opt for "Seattle".

```
wp_page = wp.WikipediaPage("Seattle")
```

We saved the page in the variable `wp_page`, to which we can add attribute commands to retrieve information, such as the URL, categories, images, sections, etc. We can retrieve the entire HTML content by adding `html()`.

```
wp_page.html()
```

```
'<div class="mw-content-ltr mw-parser-output" lang="en"
dir="ltr"><div class="shortdescription […]'
```

However, we cannot read such an output well. We can import and use the HTML package to view the content on the webpage.

```
from IPython.core.display import HTML

HTML(wp_page.html())
```

Let us have a look at page attributes.

```
wp_page.title, wp_page.url
```

```
('Seattle', 'https://en.wikipedia.org/wiki/Seattle')
```

Similarly, we can use `content`.

```
wp_page.content
```

```
'Seattle (see-AT-əl) is a seaport city on the West Coast of the
United States. It is the […]'
```

Or we can look at `references`, which will retrieve the URLs of the external links of the page.

```
wp_page.references
```

```
['http://www.seattleopera.org/discover/wagner/index.aspx',
 'http://www.playbillarts.com/news/article/5090.html',
 […] ]
```

We can access the images on the page using `images`. Let us retrieve the first three images.

```
wp_page.images[:3]
```

```
['https://upload.wikimedia.org/wikipedia/commons/a/aa/Abies_las
iocarpa_0775.JPG',
'https://upload.wikimedia.org/wikipedia/commons/2/2f/Amazon_Sph
eres_from_6th_Avenue%2C_April_2020.jpg',
'https://upload.wikimedia.org/wikipedia/commons/9/98/Ambox_curr
ent_red.svg']
```

A handy function within Python is `dir()`, which shows the **dir**ectory (list) of all attributes of an object (including functions). If we use it on our variable `wp_page`, we can see that it lists all the attributes also specified in the documentation (and a few more internal functions). Using `dir()` can help us to get a quick overview of what we can do with the current object.

```
dir(wp_page)
```

```
[ […], 'html', 'images', 'links', […] ]
```

Next, we can save the information in a more structured format to make it easier to manipulate the data further. Let us save some of the information in a pandas DataFrame.

```
import pandas as pd
data = pd.DataFrame(
data = [[wp_page.title, wp_page.url, wp_page.content,
wp_page.images, wp_page.references, wp_page.links,
wp_page.categories]],
columns = ['Title', 'URL', 'Content', 'Images', 'References',
'Links', 'Categories'])
data.head()
```

```
[Title    URL                                        Content […]
Seattle  https://en.wikipedia.org/wiki/Seattle      Seattle […]
```

A row represents a Wikipedia page, and the columns are the associated attributes of that page. We can also change the language of the Wikipedia pages through the `set_lang()` function. We need to keep in mind to search for page titles in the language we have set.

```
wp.set_lang("es") # spanish
wp.summary("Seattle", sentences=2)
```

```
'Seattle (/siːˈæ|əɫ/) es la ciudad más grande del estado de […]'
```

Let us set it back to English.

```
wp.set_lang("en")
```

1.2 Retrieving tables

In some cases, we might be interested in content found in tables on Wikipedia pages. For example, a list of Nobel laureates or a list of political parties in Germany. Retrieving table content can be tricky since Wikipedia pages can contain many markups (i.e., indicators of links, formats, etc.) that can obscure the content we are interested in.

Using the Wikipedia page "List of Nobel laureates" as an example, we want to extract the data in the table on that page. First, we can load the page with the package `wikipedia` using the `page()` function.

```
page_nobel = wp.page("List of Nobel laureates")
```

Next, we want to save the page in an HTML format with the `html` attribute. We also need to specify how the page should be encoded (i.e., how the function should understand the wiki code). In this case, we select UTF-8, a standard format that allows us to obtain the content in the table in a human-readable form (for more information on character encoding, see here).

```
html_nobel = page_nobel.html().encode("UTF-8")
```

We can now save the HTML page in a pandas DataFrame. Here, we also need to specify the encoding format. Since the `read_html()` function saves all the page information as tables, a list of tables is created, and we need to specify which table we are interested in. In this case, the first table, indicated with `[0]`.

```
nobel_table = pd.read_html(html_nobel, encoding="utf-8",
    index_col=0)[0]
nobel_table.head()
```

The output is shown in **Table 1**.

Year	Physics	Chemistry	Physiology or Medicine
1901	Wilhelm Röntgen	Jacobus Henricus van 't Hoff	Emil von Behring
1902	Hendrik Lorentz; Pieter Zeeman	Emil Fischer	Ronald Ross
...	...	...	...

Table 1. Nobel laureates

1.3 Retrieving Wikipedia page revisions

Wikipedia pages are constantly edited and discussed among Wikipedians. Since page changes are tracked, we can access past versions and revisions. We will look at how we can extract revisions of Wikipedia pages using the packages Pywikibot and MWParserFromHell.

- Pywikibot is a wrapper that provides us with the markup code of Wikipedia pages (i.e., indicators for links, tables, paragraphs, etc.).
- MWParserFromHell is a parser that can read the markup and help us obtain the information we are interested in.

Before we can import these packages into our environment, we need to install them. Both packages are unavailable through Anaconda but can be installed with pip.[1]

```
import pywikibot
import mwparserfromhell
```

Using the `Site()` and `Page()` function from `pywikibot`, we can load the Wikipedia website and the page we are interested in.

```
site = pywikibot.Site('en', 'wikipedia')
page = pywikibot.Page(site, "Seattle")
```

We can then obtain all the page revisions using the `revisions()` attribute. Depending on the number of revisions, it can take a few seconds. Since the Wikipedia entry for Seattle has many revisions, we will limit the retrieved revisions to 200 with `total` (it might take a few seconds).

```
revisions = page.revisions(content=True, total=200)
```

We can list the revisions and group them in the year they have been written. Each revision is a dictionary, and we can retrieve the years using the `timestamp` key in those dictionaries.

```
revisions_list = []
years = []

for revision in revisions:
    revisions_list.append(revision)
    years.append(int(str(revision["timestamp"])[:4]))
print(years)
```

```
[2024, 2024, 2024, 2024, 2024, 2024, 2024, […], 2023]
```

[1] If possible, it should be avoided to use conda and pip simultaneously to handle packages. If Anaconda is used, but a package is not available, the best option is to create a virtual environment, in which pip is used.

We can see that the last 200 revisions reach back to 2023.

Using the `keys()` function, we can look at the type of information the revision dictionary contains without looking at the content.

```
print(list(revisions_list[0].keys()))
```

```
['revid', 'parentid', 'user', 'userid', 'timestamp', 'size',
'sha1', 'roles', 'slots', 'comment', 'parsedcomment', 'tags',
'anon', 'minor', 'userhidden', 'commenthidden', 'text',
'contentmodel']
```

In many cases, we are interested in the actual text of the revision and the user who made that revision. Let us have a look at what each entry looks like.

```
revisions_list[0]["user"]
```

```
'William Avery'
```

```
revisions_list[0]["text"]
```

```
'{{short description|Largest city in Washington,
U.S.}}\n{{about|the city}}\n{{pp-move}}\n{{pp-semi-
indef|small=yes}}\n{{Use mdy dates|date=February 2024}}\n […]'
```

The username is readable, but the text contains many markups that make it difficult to read or process further for text analyses. We can parse the text with `mwparserfromhell` to make the text more human-readable.

```
parsed_text = mwparserfromhell.parse(revisions_list[0]["text"])
print(parsed_text.strip_code())
```

```
Seattle ( ) is a seaport city on the West Coast of the United
States. It is the seat of King County, Washington. With a 2022
population of 749,256 it is the most populous city in both the
state of […]
```

Looks better! We can now iterate over the text and save the year, user, and cleaned text. We can define a function that cleans our text to simplify our code.

```
def clean_text(text):
    parsed_text = mwparserfromhell.parse(text)
    return parsed_text.strip_code()

# Obtain revisions - might take a few seconds.
revisions = page.revisions(content=True, total=200)
```

```
# Initiate lists
years = []
users = []
texts = []

# Iteration
for revision in revisions:
    years.append(int(str(revision["timestamp"])[:4]))
    users.append(revision["user"])
    texts.append(clean_text(revision["text"]))
```

Finally, we can save the data into a DataFrame, making future analyses easier.

```
The output revision_data = pd.DataFrame({'year': years, 'user':
users, 'text': texts})

revision_data.head()
```

The output is shown as **Table 2**.

year	user	text
2024	William Avery	Seattle () is a seaport city on the West Coas...
2024	Cfls	Seattle () is a seaport city on the West Coas...
2024	SounderBruce	Seattle () is a seaport city on the West Coas...
2024	Bgarrott2023	Seattle () is the most populous city in, and ...
2024	Citation bot	Seattle () is a seaport city on the West Coas...

Table 2. Revisions of “Seattle” page

Further text processing is out of the scope of this guide, and we will end here with the coding instructions. However, additional material about data wrangling, cleaning, and visualization can be found in this list of resources, as well as in this GitHub repository with useful Jupyter Notebooks.

4 Conclusion

This guide provides an overview on how to work with APIs through wrappers, with Wikipedia as an example. We demonstrated how to query the Wikipedia API to obtain data from articles. By utilizing various Python packages, we showed how to save the obtained data in text and table form, making it suitable for further (text) analyses. Thus, the guide provides the basics for working with APIs as the principle of querying, retrieving, and managing data remains largely similar across APIs. Once those principles are understood, the remaining difficulties lie in learning and using the individual commands specific to the API, that are, ideally, documented. In many cases, more than one wrapper exists for an API, often catering to specific needs (e.g., analyzing Wikipedia revisions), and a quick web search including the data type or research goal will help find them. Since APIs change, wrappers must adapt and might become outdated if not maintained. Checking the latest updates when finding a wrapper can indicate if it still works when implemented. Similarly, before using an API or wrapper, their latest documentation should be read, which will help determine if they are suited for the intended purpose.

References

Abouelhassen, M. A. (2024, January 23). Introducing Facebook Graph API v19.0 and Marketing API v19.0. *Facebook Developers Blog*. https://developers.facebook.com/blog/post/2024/01/23/introducing-facebook-graph-and-marketing-api-v19/

Google for Developers. (2022, April 11). *YouTube data API - quota and compliance audits*. https://developers.google.com/youtube/v3/guides/quota_and_compliance_audits

Huffman, S. [spez]. (2023, June 19). *Addressing the community about changes to our API* [Online forum post]. Reddit. https://www.reddit.com/r/reddit/comments/145bram/addressing_the_community_about_changes_to_our_api/

Klawonn, T. (2020, January 7). Grenzen des „Web Scraping". *Forschung & Lehre*. https://www.forschung-und-lehre.de/recht/grenzen-des-web-scrapings-2421/

Perriam, J., Birbak, A., & Freeman, A. (2020). Digital methods in a post-API environment. *International Journal of Social Research Methodology*, *23*(3), 277-290. https://doi.org/10.1080/13645579.2019.1682840

Reddy, M. (2011). *API Design for C++*. Elsevier. https://doi.org/10.1016/C2010-0-65832-9

Soldner, F. (2024). *Overview of approaches for collecting data from online platforms* (GESIS Guides to Digital Behavioral Data, 8). Cologne: GESIS – Leibniz Institute for the Social Sciences.

All references and links were retrieved on July 15, 2024.

Acknowledgements

The authors are grateful to Roberto Ulloa for contributing to figure 1.

About the author(s)

Felix Soldner is a post-doctoral researcher at GESIS – Leibniz Institute for the Social Sciences in Cologne, Germany at the Computational Social Science department. He works with data science methods, such as Machine Learning (ML) and Natural Language Processing (NLP), to research subjects, such as deception detection, fraud prevention, dark web markets, and data biases impacting ML performances. For his projects he uses various data collection approaches, including custom scrapers and APIs.

N. Gizem Bacaksizlar Turbic is a post-doctoral researcher at GESIS – Leibniz Institute for the Social Sciences in Cologne, Germany at the Computational Social Science department. Her research areas include social and political networks, and social media analysis.

Pouria Mirelmi is a master's student in Computational Social Science at RWTH Aachen University, and a student assistant at GESIS – Leibniz Institute for the Social Sciences. His current research focus is Social Network Analysis and Large Language Models.

Julian Kohne is a doctoral researcher in the team Designed Digital Data in the Department of Computational Social Science at GESIS – Leibniz Institute for the Social Sciences in Cologne, Germany, and the Department of Molecular Psychology at Ulm University, Germany. His work at GESIS contributes to developing an app for collecting survey data and digital behavioral data using smartphones. In his dissertation at Ulm University, he is using donated WhatsApp chat log data to investigate communication in close interpersonal relationships. More information: https://www.juliankohne.com/

Suggested citation

Soldner, F., Bacaksizlar Turbic, N. G., Mirelmi, P., & Kohne, J. (2024). *How to collect web data via APIs* (GESIS Guides to Digital Behavioral Data, 9). Cologne: GESIS – Leibniz Institute for the Social Sciences.

Series editors

Danica Radovanović, Maria Zens, Johannes Breuer, Katrin Weller, Claudia Wagner

Publisher

GESIS Leibniz Institute for the Social Sciences

License

Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0 Deed)