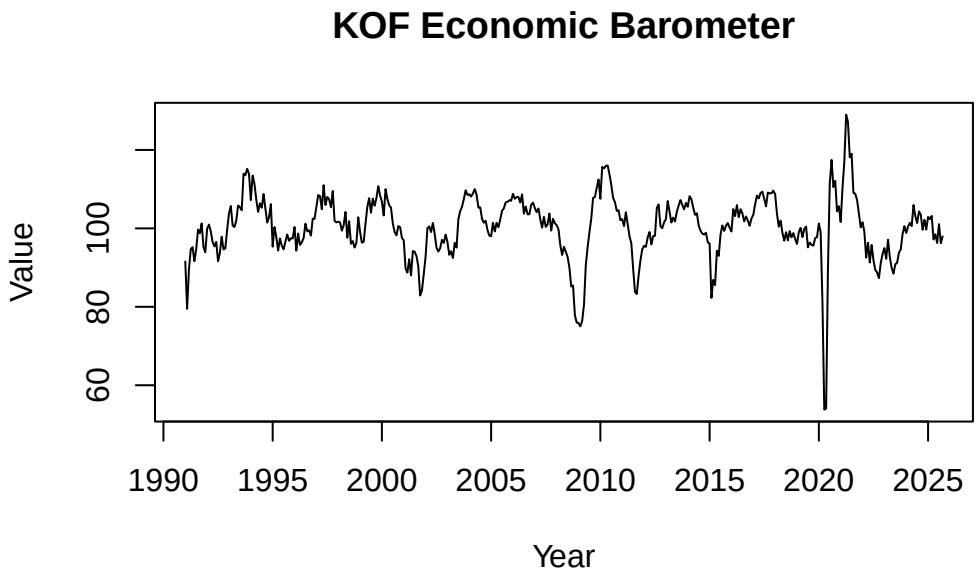
Guest Lecture 2025 - Leverage APIs for Economic Data

Minna Heim

What you see in the plot below is the KOF Economic Barometer, which is responsible for predicting how the Swiss economy should perform in the near future Source. To do this well, the Barometer, and other Economic Indices include other time series data, which have an economically plausible influence on the Swiss business cycle. In the case of the KOF Barometer, this is upwards of 300 different variables, so upwards of 300 different time series, all to capture the complex dynamics of the Swiss Economy into one number.



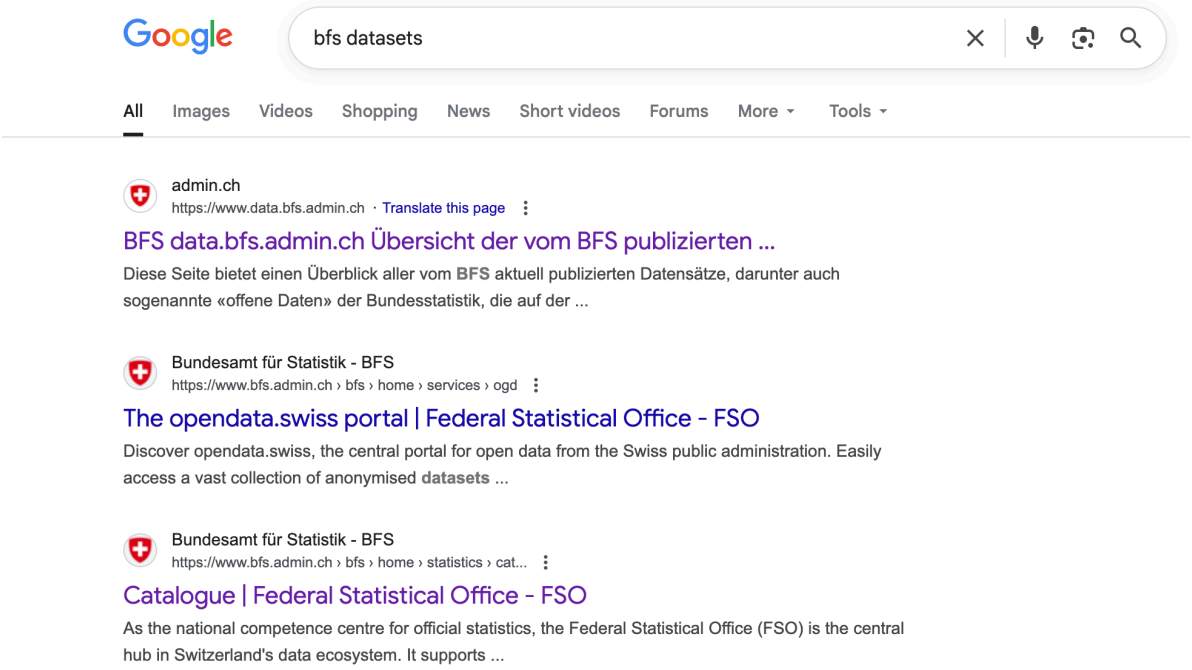
The reason why I am presenting this Economic Index to you, is because although this blog post is about APIs and their usage for economic data, we will go through the beginning of what it takes to create our own Economic Index - namely what it takes to efficiently get upwards of 300 time series.

By the end of this blog post, you will gain a better understanding of API's and their usage, but moreover, you will have built your own API wrapper, and even potentially, your own API. All of this will hopefully help you identify APIs in the real world - and realize, that APIs are used everywhere - from Art Museums to COVID-19 case rates.

Getting Data, the Naive Way:

Let's go back to our initial example: when trying to gather the data to build our economic Index, how would we do this?

The Naive way of doing this, would be to google the data we need from the Swiss Federal Statistical Office (BFS), such as Swiss GDP- which is a good representation of economic activity, but not the only one - and see what we can find.



Then, by clicking onto the BFS' Website, we could find their Economic Data Portal.

Übersicht der vom BFS publizierten Datensätze

Diese Seite bietet einen Überblick aller vom BFS aktuell publizierten Datensätze, darunter auch sogenannte «offene Daten» der Bundesstatistik, die auf der OGD-Plattform des Bundes opendata.swiss zugänglich sind, sowie die Daten der National Summary Data Page (NSDP).

Die Datensätze sind nach den bekannten 21 statistischen Themen (www.bfs.admin.ch) und nach dem Grad der Zugänglichkeit gruppiert: Tabellen (xlsx), maschinenlesbare Daten (csv, json) und solche, die über API zugänglich sind.

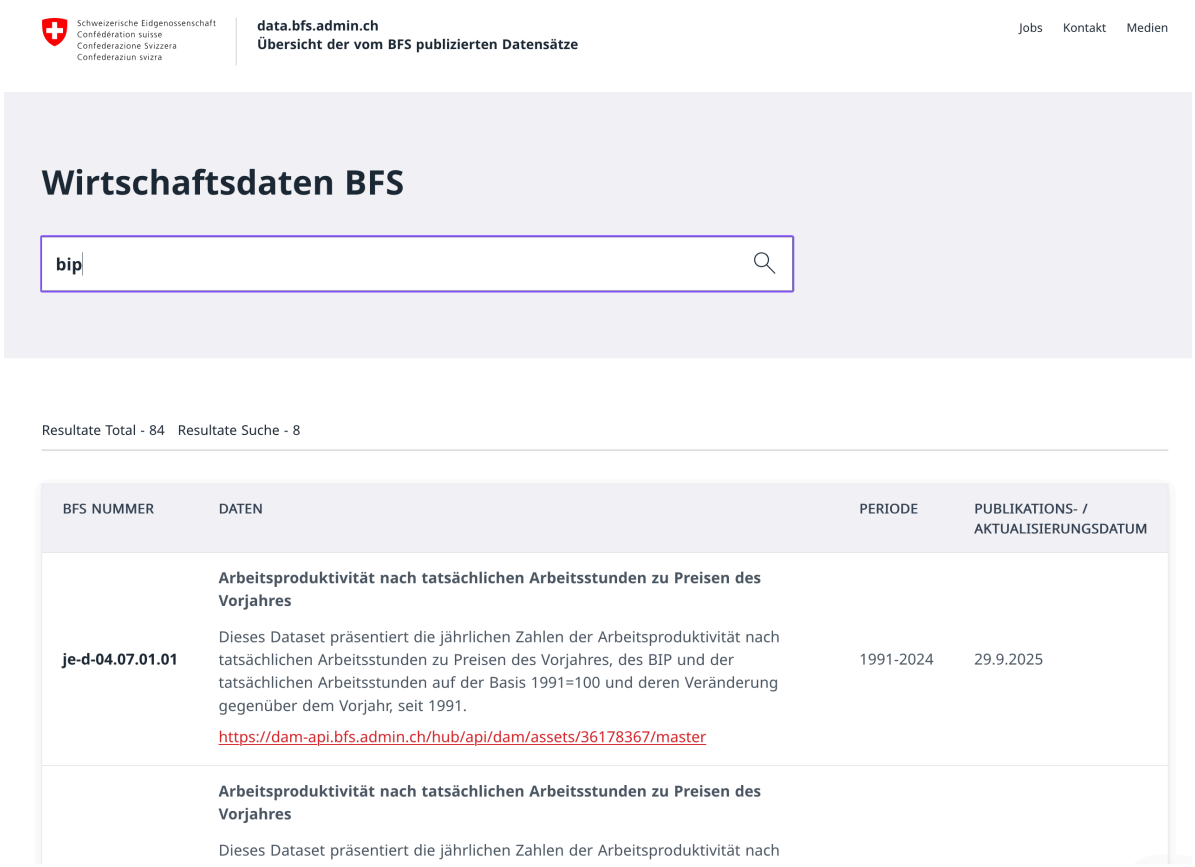
Im Fokus

BFS-Datensätze
Gleichstellung von Frauen und Männern

BFS-Datensätze
Wirtschaftsdaten BFS

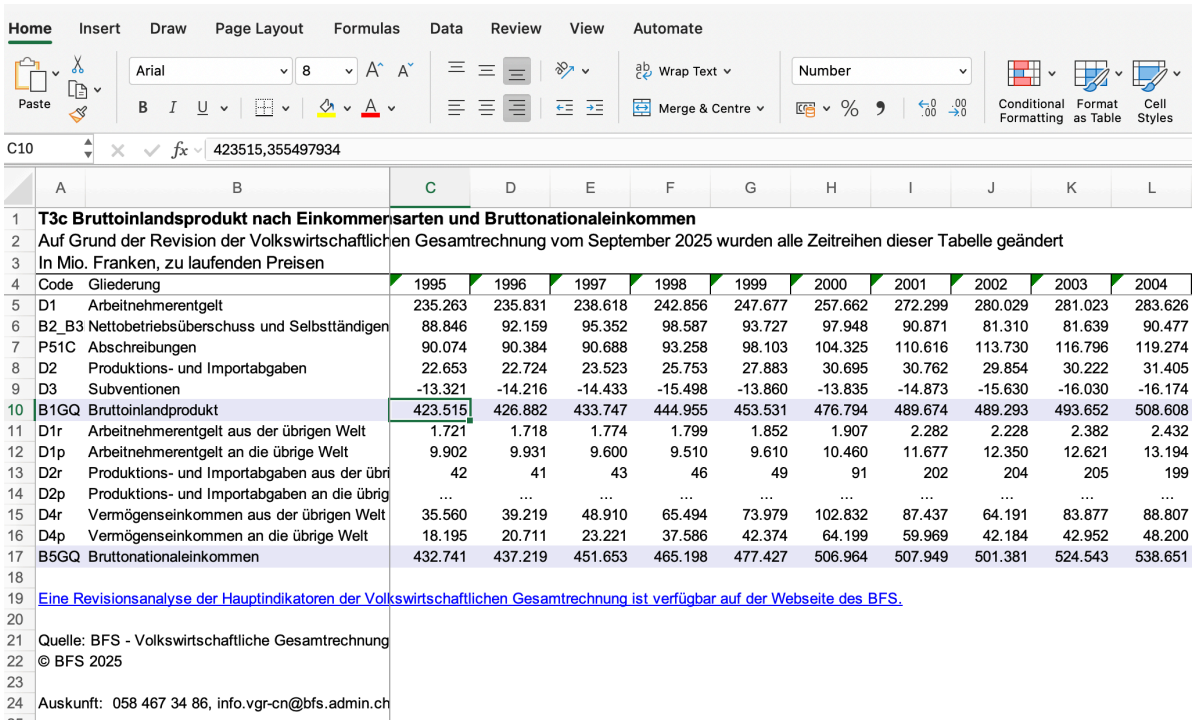
BFS-Datensätze
Statistik der Schweizer Städte 2025

Then, after inputting our desired time series, we would find different variations of gdp data.



After downloading the dataset from BFS, we can open and inspect it. The data is in excel and in wide format (meaning values are horizontally oriented) because it is more humanly readable this way, and nicer to look at.

When programming, we often prefer it to be in long format (values vertically oriented) because it makes plotting easier and because most time series data is represented this way.



Home Insert Draw Page Layout Formulas Data Review View Automate

Paste | Arial | 8 | Wrap Text | Number | Merge & Centre | Conditional Formatting | Format as Table | Cell Styles

C10 | 423515,355497934

	A	B	C	D	E	F	G	H	I	J	K	L
1	**T3c Bruttoinlandsprodukt nach Einkommensarten und Bruttonationaleinkommen**											
2	Auf Grund der Revision der Volkswirtschaftlichen Gesamtrechnung vom September 2025 wurden alle Zeitreihen dieser Tabelle geändert											
3	In Mio. Franken, zu laufenden Preisen											
4	Code	Gliederung	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004
5	D1	Arbeitnehmerentgelt	235.263	235.831	238.618	242.856	247.677	257.662	272.299	280.029	281.023	283.626
6	B2_B3	Nettobetriebsüberschuss und Selbsttändigen	88.846	92.159	95.352	98.587	93.727	97.948	90.871	81.310	81.639	90.477
7	P51C	Abschreibungen	90.074	90.384	90.688	93.258	98.103	104.325	110.616	113.730	116.796	119.274
8	D2	Produktions- und Importabgaben	22.653	22.724	23.523	25.753	27.883	30.695	30.762	29.854	30.222	31.405
9	D3	Subventionen	-13.321	-14.216	-14.433	-15.498	-13.860	-13.835	-14.873	-15.630	-16.030	-16.174
10	B1GQ	Bruttoinlandprodukt	423.515	426.882	433.747	444.955	453.531	476.794	489.674	489.293	493.652	508.608
11	D1r	Arbeitnehmerentgelt aus der übrigen Welt	1.721	1.718	1.774	1.799	1.852	1.907	2.282	2.228	2.382	2.432
12	D1p	Arbeitnehmerentgelt an die übrige Welt	9.902	9.931	9.600	9.510	9.610	10.460	11.677	12.350	12.621	13.194
13	D2r	Produktions- und Importabgaben aus der übri	42	41	43	46	49	91	202	204	205	199
14	D2p	Produktions- und Importabgaben an die übrig	...	...	...	...	...	...	...	...	...	...
15	D4r	Vermögenseinkommen aus der übrigen Welt	35.560	39.219	48.910	65.494	73.979	102.832	87.437	64.191	83.877	88.807
16	D4p	Vermögenseinkommen an die übrige Welt	18.195	20.711	23.221	37.586	42.374	64.199	59.969	42.184	42.952	48.200
17	B5GQ	Bruttonationaleinkommen	432.741	437.219	451.653	465.198	477.427	506.964	507.949	501.381	524.543	538.651
18												
19	Eine Revisionsanalyse der Hauptindikatoren der Volkswirtschaftlichen Gesamtrechnung ist verfügbar auf der Webseite des BFS.											
20												
21	Quelle: BFS - Volkswirtschaftliche Gesamtrechnung											
22	© BFS 2025											
23												
24	Auskunft: 058 467 34 86, info.vgr-cn@bfs.admin.ch											

So, once downloaded, we have to manipulate the data so that it fits our purpose:

```
library(readxl)
library(magrittr)
library(tidyr)

data <- read_xlsx("examples/je-d-04.02.01.03.xlsx")

# subset only the rows 3 (= year) and 9 (= gdp)
gdp <- data[9,]
names(gdp) <- data[3,]

# pivot data longer (from excel wide format to data frame long format)
gdp_long <- gdp %>%
  pivot_longer(
    cols = everything(),
    names_to = "Year",
    values_to = "GDP"
  )

# remove old headers
```

```r
gdp_long <- gdp_long[-c(1:2),]

# check structure
# str(gdp_long)

# convert to gdp numeric and year to year
gdp_long$GDP <- as.numeric(gdp_long$GDP)
gdp_long$Year <- as.integer(gdp_long$Year)

head(gdp_long, n=10)
```

We read in our data, then select the rows which we need, which is the aggregated GDP, and the date. Then we pivot our data (which transforms the data from horizontally oriented to vertically oriented), and remove old headers. Finally, we make sure our variables are represented in their correct format. Then, after all of this, our data looks like this:

```
# A tibble: 10 x 2
    Year     GDP
   <int>   <dbl>
 1  1995 423515.
 2  1996 426882.
 3  1997 433747.
 4  1998 444955.
 5  1999 453531.
 6  2000 476794.
 7  2001 489674.
 8  2002 489293.
 9  2003 493652.
10  2004 508608.
```

Finally, we have successfully searched for, imported and cleaned our first time series dataset for our Economic Index. It was quite a long and intensive process, wasn't it? We had to look for the correct data, and then clean it all. And given that we have approx. 299 time series to go, this could become quite unruly & complicated. - Especially if all of a sudden, it's a new year and we want to include the newest data into our analysis - then we have to repeat the entire process!

Especially because GDP is quite the common dataset, it's made readily available and published in a well known format, like excel. But once we get to more niche time series, they could be not only more difficult to find, but also more difficult to import and clean.

Because of this, let's introduce a different method for extracting data from a public source - that is more reproducible, and we want to do it using code.

Getting Data, the API (wrapper) Way:

API stands for Application Programming Interface, and essentially standardizes the way in which your computer communicates with another computer.

I've made the statement before, that the "naive way" of doing things becomes unruly & complicated - the reason why APIs are preferred over this method is because when using APIs everything is standardized -> because of protocols.

A Protocol - for those of you not familiar - are just explicit set of rules which need to be followed. APIs use something called HTTP - which dictates how all data transfer happens over the web.

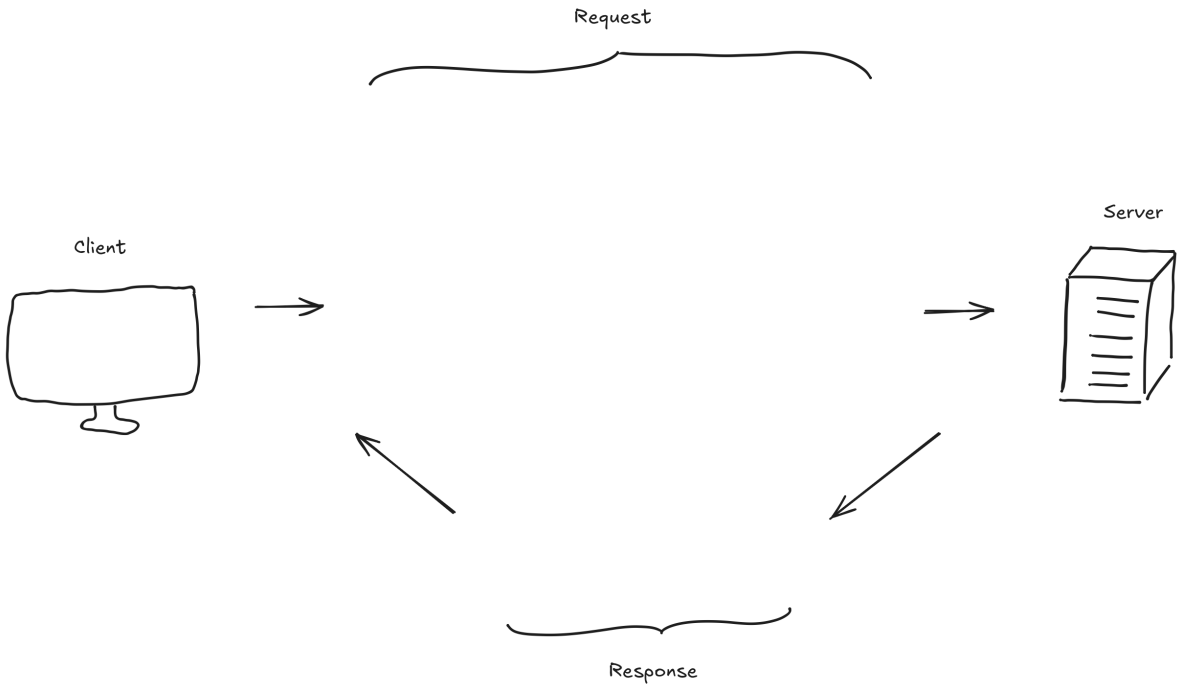
You might have already seen HTTP in your browser (e.g. http://url...) this is because your browser on your computer and the server also communicate via HTTP.

But just communicating via http:// in our browser is not enough, because then this is still not reproducable, we need to input the urls every time.

We want to find a solution that is reproducible & which uses code - aka we want our R code to communicate with the server - so we use APIs.

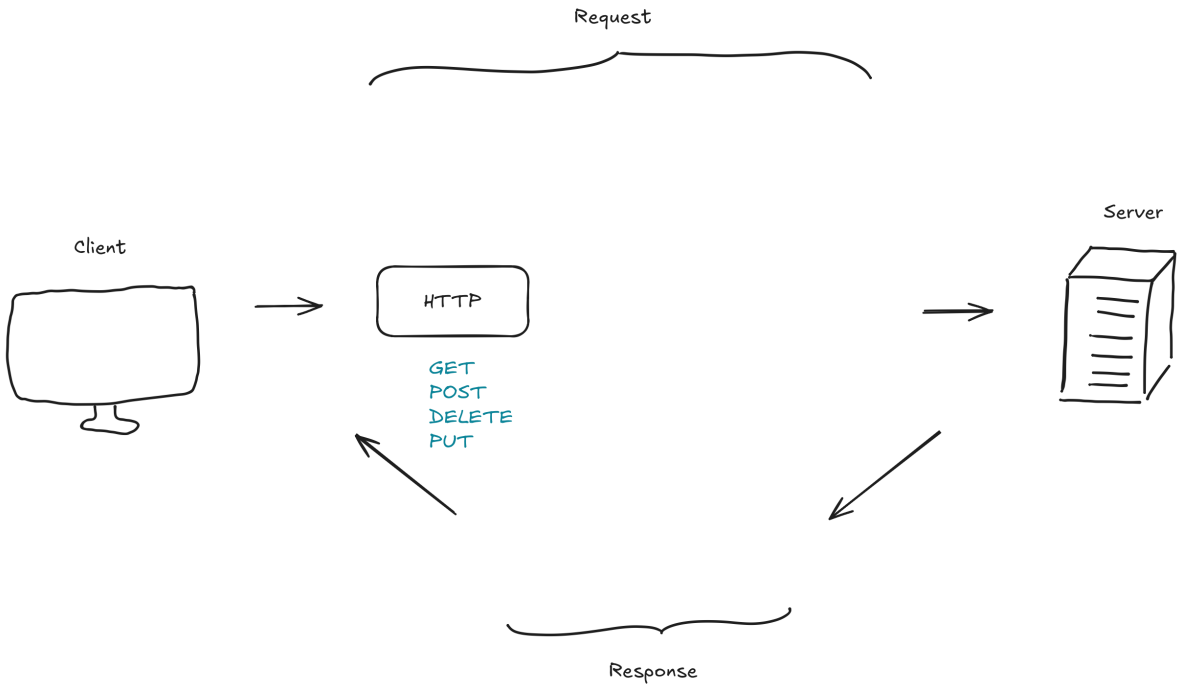
There are 2 main important things I want you to know about HTTP:

it uses *client & server* and *request & response* protocols (aka clients request something from the server, the server sends a response)



And that there are 4 main **HTTP request methods** used to determine the type of data exchange happening:

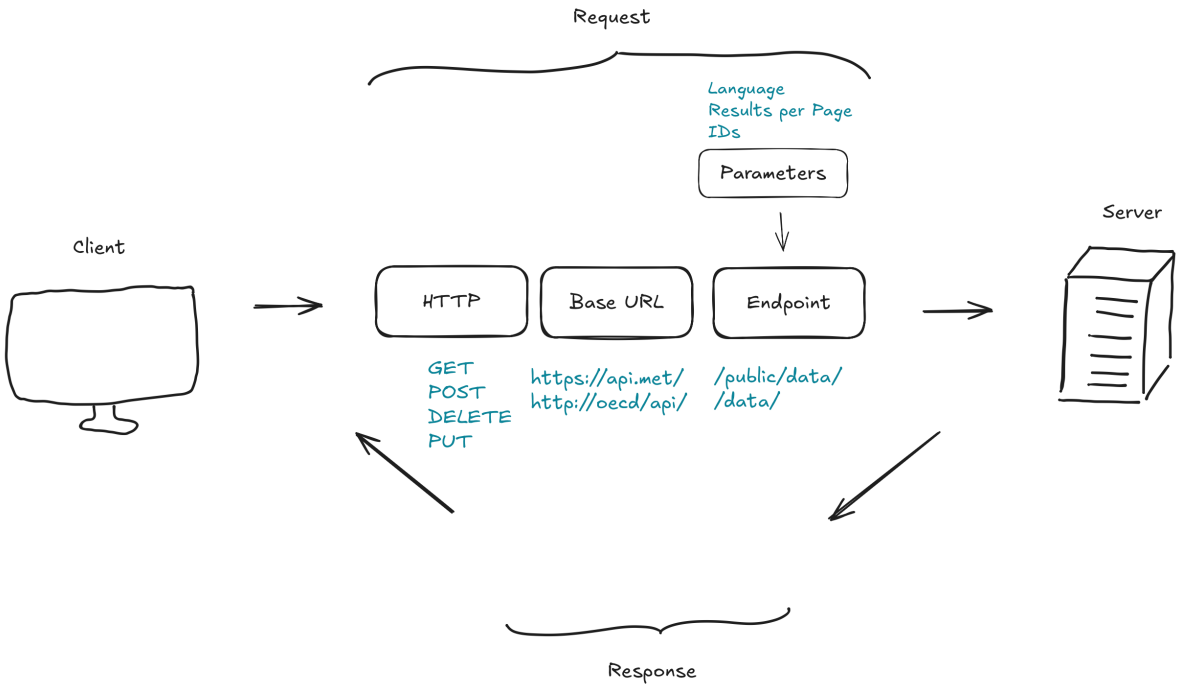
- GET : retrieves data
- POST : sends data
- PUT: changes the data
- DELETE : deletes data



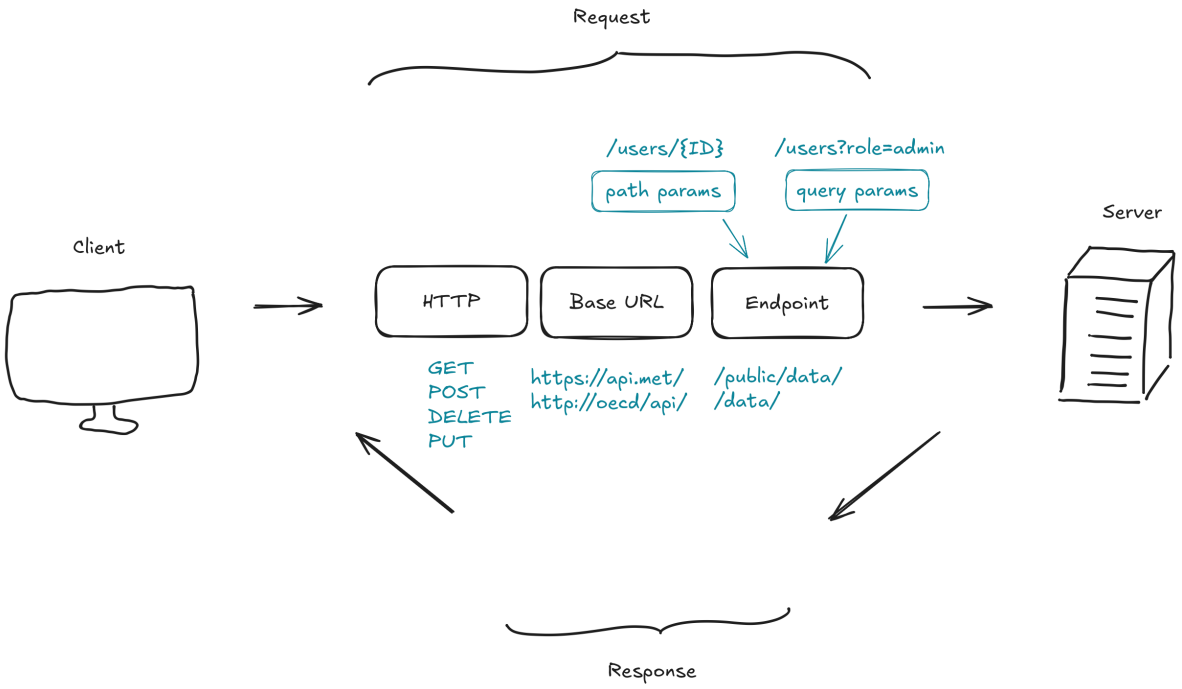
Now that we know that APIs use the HTTP protocol, let's put it all together to explain what an API looks like.

The **request** consists of:

- one of the HTTP request methods, such as GET (to get data) and
- the **Where** which is Base URL, since it represents the location of the server.
- the **Which service** is specified through the Endpoint, which dictates whether the client wants all data, just the data with a specific ID, the user with a specific ID, etc.

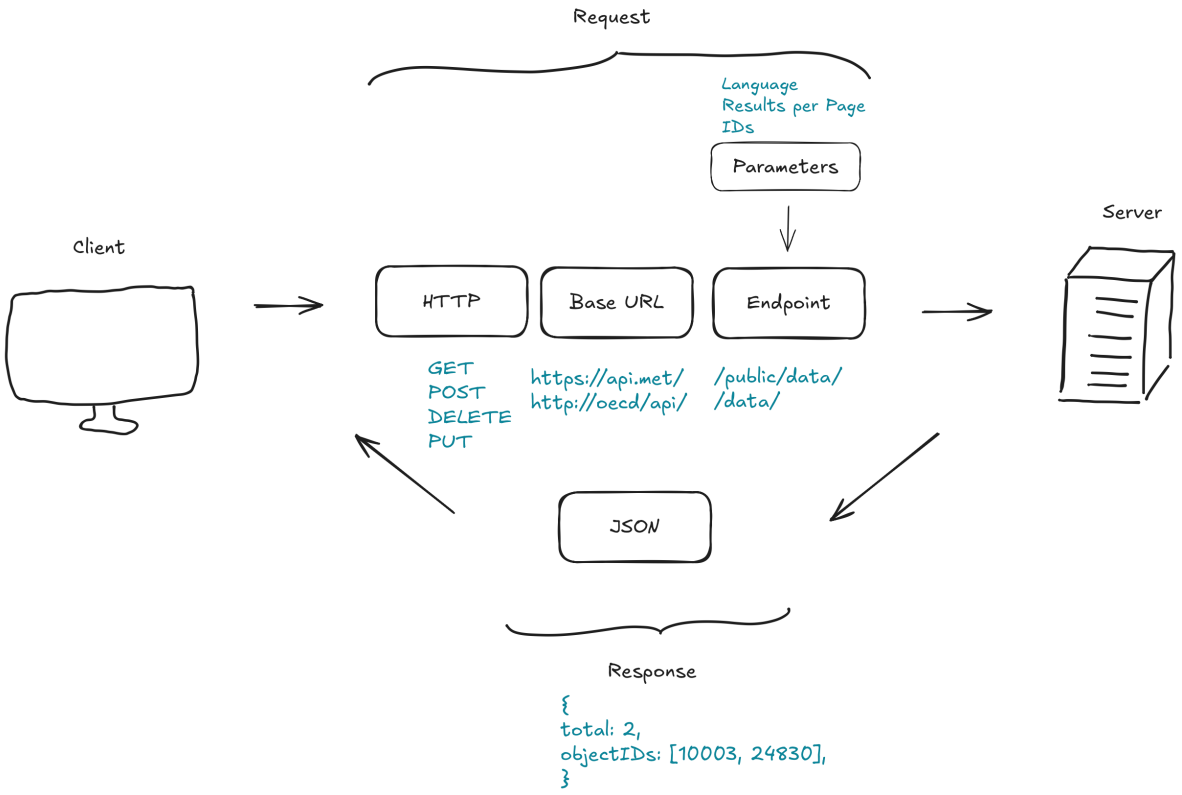


- then the **how** is specified with additional parameters. You can think of them like search filters. There are 2 types of parameters:
  - *path parameters*, like `/users/{userID}` or
  - *request body parameters* like `/users?ID=userID` (& can also be used to separate multiple body params)

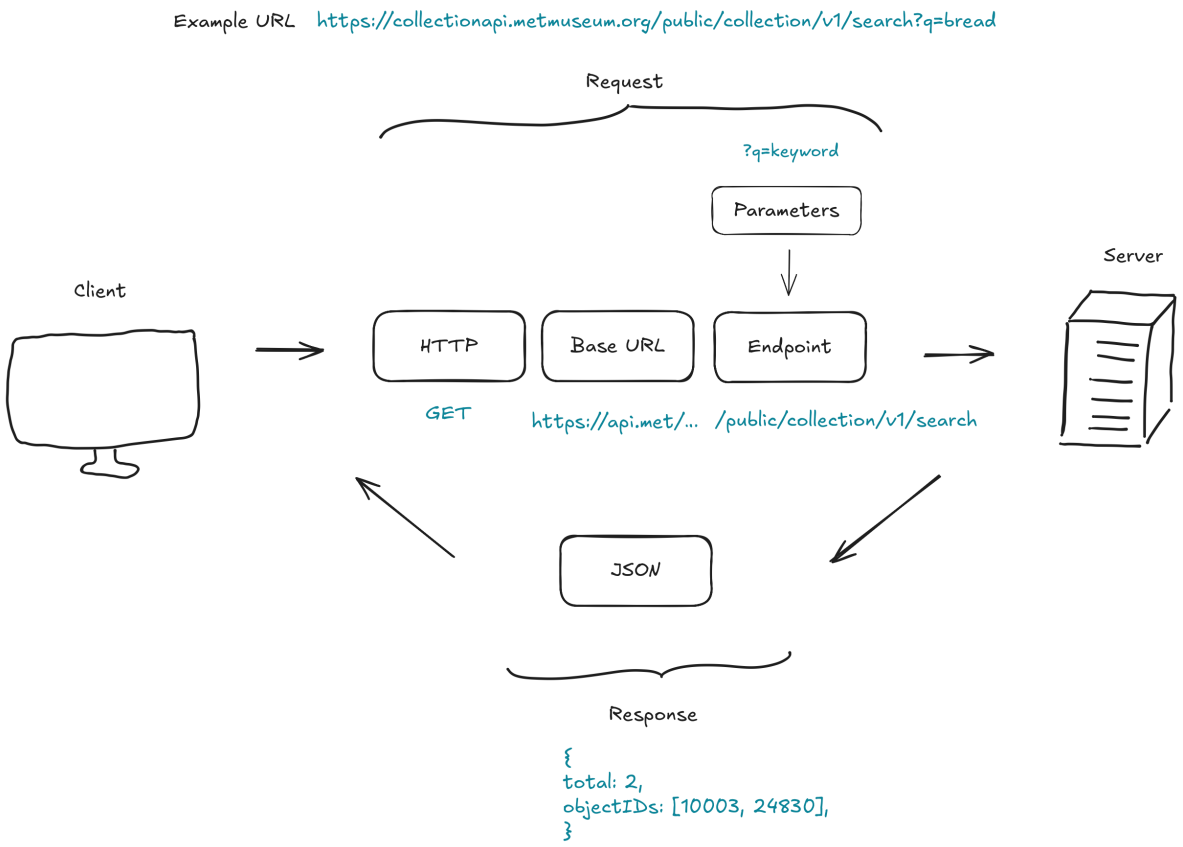


Once this request is sent to the server, it gives back a **response**:

- **Status**: if the request was successful or unsuccessful
- **Response Body**: depending on the request you might get requested data, an acknowledgment that a resource was created or updated, an error message. Common formats of the response body are JSON and HTML.



Let's use what we've learnt to look at an example:



Here, we're using the GET request method, the base URL of the met museum, the endpoint to the museum collections and to the search endpoint, and adding a keyword as a parameter to get the number of objects and IDs of objects which contain this keyword in their title.

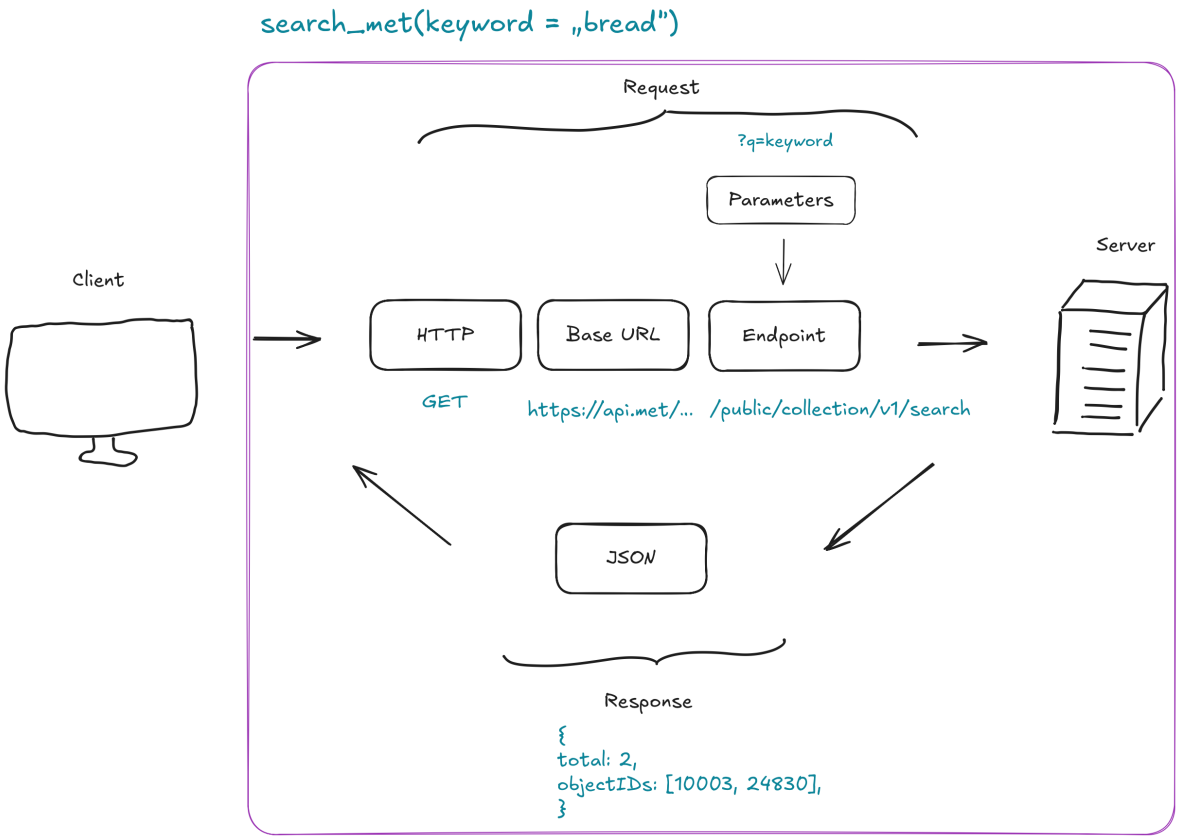
I encourage you to try out GET API call yourself, by putting in the URL of the Example into your browser and inputting your own search word. Since HTTP is used on the web, we can perform the GET request via our browser as well.

URL: https://collectionapi.metmuseum.org/public/collection/v1/search?q=bread

API Wrappers

The most frequent way we as programmers encounter APIs is through so called API wrappers, you can think of them like simple programs that are intermediaries between you and your API, so they wrap the API.

API wrappers usually allow you to do complex API calls in an easier environment and are tailored to your programming language, so usually easier for us to use.



Here is a visual example of how you might encounter the previous example in an API wrapper. Notice how the functionality stays the same, we can perform the same tasks, but see and interact with less of the complexity **(=information hiding)**

Use the BFS API Wrapper for R

Let's go back to our example of getting data from the Swiss Federal Statistical Office (BFS). Since we have already imported and cleaned our gdp data, we want to move on to the next relevant time series. One other factor that could be a good predictor for economic activity in Switzerland is energy consumption over time.

Luckily for us, we can use the BFS' API wrapper, which is the BFS R Package to get our energy data.

```
# devtools::install_github("lgnbhl/BFS")
library(BFS)

# to inspect functions, use ?function, e.g.
```

```r
?bfs_get_catalog_data

bfs_get_catalog_data(language = "en", title = "energy")
dataset_nr <- "px-x-0204000000_106"
energy_df <- bfs_get_data(dataset_nr, lang = "en")

# filter only total energy accounts
total_energy <- subset(
  energy_df,
  # filter for totals for each of the columns
  `Economy and households` == "Economy and households - Total" &
  `Energy product` == "Energy product - Total",
  # select only the two relevant columns
  select = c(Year, `Energy accounts of economy and households`)
)

names(total_energy)[2] <- "Amount"

# View(total_energy)
head(total_energy, n=10)
```

Let's go through the code line-by-line:

First we install and load the package, then we can inspect the functions we will use, such as `bfs_get_catalog_data` to see that we can use this function to determine what kind of data BFS publishes, based on keywords.

```
# A tibble: 1 x 6
  title                language number_bfs number_asset publication_date url_px
  <chr>                <chr>    <chr>      <chr>        <date>           <chr>
1 Energy accounts of e~ en       px-x-0204~ 36179320     2025-09-30       https~
```

We search for energy datasets, save the respective ID of the dataset, and then GET the data, using the `bfs_get_data` function.

After line 9, we are already done with the API usage for this example. The rest is just filtering the dataset to get only the columns of interest (total Energy Product across all Economy and Households)

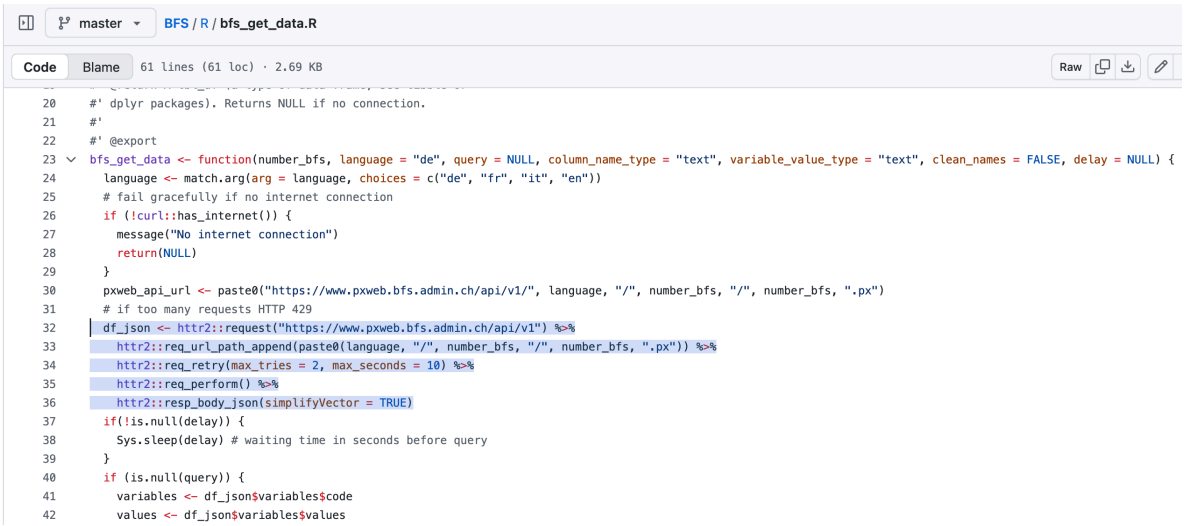
When inspecting the data, it looks like this:

```
# A tibble: 10 x 2
   Year    Amount
   <chr>    <dbl>
 1 2000  1103603.
 2 2001  1127768.
 3 2002  1112879.
 4 2003  1130842.
 5 2004  1131767.
 6 2005  1121999.
 7 2006  1157170.
 8 2007  1126804.
 9 2008  1158734.
10 2009  1136915.
```

From API Wrappers to APIs proper

In theory, we could be done with this example - using API wrappers instead of manually looking for data on the web makes it easier for us to scale our example - meaning we can easily fetch our 300 time series, with little extra effort.

Still, to get a deeper understanding of how APIs work, we could also take a look at how API Wrappers work under the hood. Luckily for us, R packages like BFS are open source (like the R programming language) so we can see how a function such as `bfs_get_data` is written. We can go to the Package author's github page, and find the function



This might look a bit complex, so let's break it down by looking at the essential code:

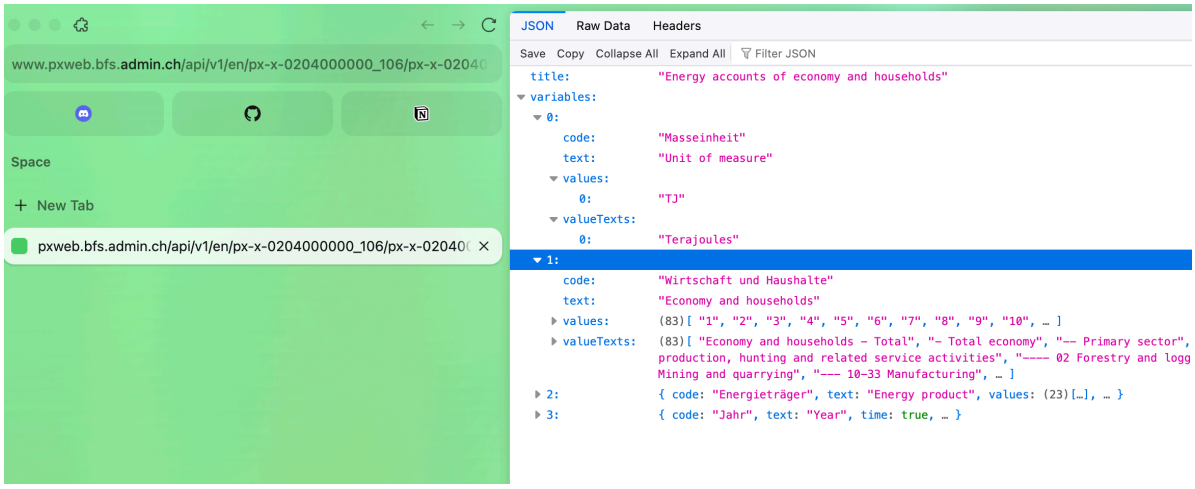
```r
bfs_get_data <- function(number_bfs, language = "de", query = NULL, column_name_type = "text"
  # base url
  df_json <- httr2::request("https://www.pxweb.bfs.admin.ch/api/v1") %>%
    # add endpoints
    httr2::req_url_path_append(paste0(language, "/", number_bfs, "/", number_bfs, ".px")) %>%
    httr2::req_retry(max_tries = 2, max_seconds = 10) %>%
    # executre request
    httr2::req_perform() %>%
    # transform response to json
    httr2::resp_body_json(simplifyVector = TRUE)
```

In essence, the function uses the existing API to build a request, then perform it, and save the response body.

Here, we build the request with the base URL, then append this URL with our endpoints which are the dataset ID and the language, and then perform the request and specify that the response body (which contains the data) should be in JSON.

Important to mention here, is that these above functions are used to fetch the general structure of the data, i.e. the Units, and different data categories. This information is then being used, to fetch the actual data via the px-web API which is a way of distributing data that statistical offices like to use - but is rarely used outside of these circles - hence I won't delve too deep into the matter. In principle, we could also use the functions presented above to get the data, and other API wrappers in R that you can find do so.

We can also try running the final URL which we create with the above functions (up to line 5) in our browser, as we did before, to see what the structure of our data looks like:



Why APIs?

By now you might have gathered an idea or two about why APIs are important and good to use.

Here are some additional benefits:

- Consistency: directly from the source, and up-to-date data (important for time series revisions)
- Speed and Scalability: as seen in our example, getting hundreds of time series through googling and searching is not scalable or quick
- Automation: the collection of data can be automated easily
- Easier to fetch data: some data is difficult to find and the specification of variables or certain time frames is easier to do.
- APIs are used everywhere! - Understanding how they work help you identify how data transfer happens

Exercises:

Now that APIs have been thoroughly introduced and their importance is clearer, try to build your own API wrapper.

Most commonly used APIs already have pre-built API wrappers, but it could be that they are not available in your programming language of choice, or are not maintained, or they don't exist at all. In that case, knowing how to go about building your own API wrapper could prove to be very useful.

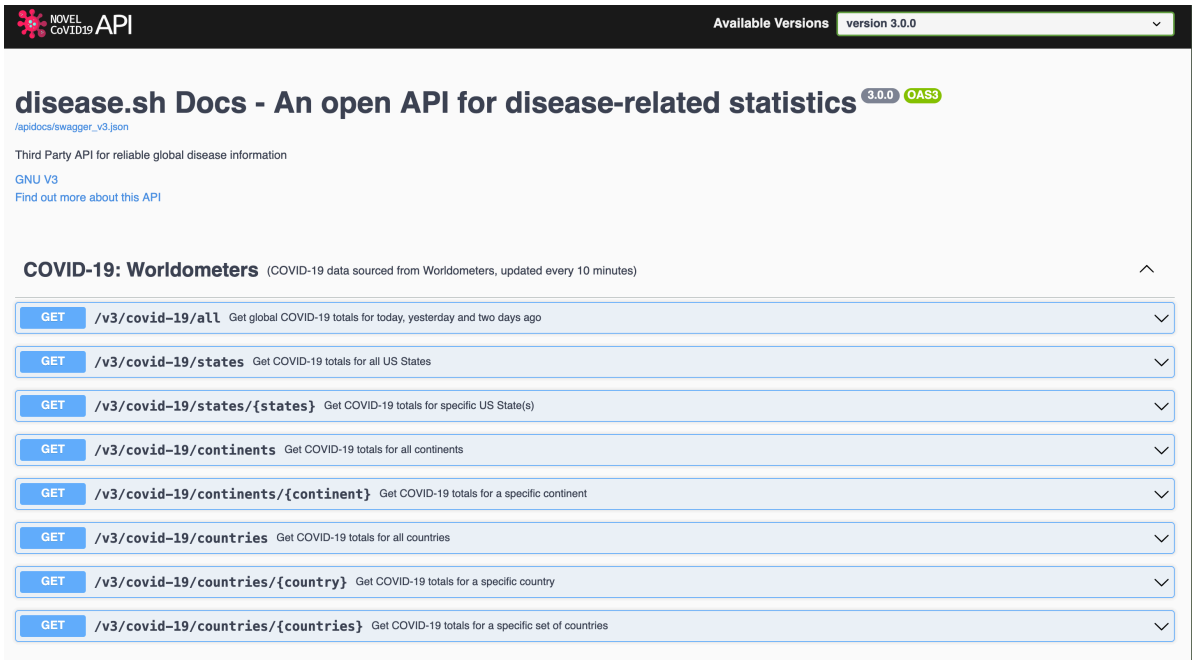
In R, the most commonly used Package to work with Web APIs is the httr2 package - on their website, you can find the documentation to get started with using httr2. A short cheatsheet summarizes the essential functions:

Category	Function	Description
Request	`request()`	Creates a new HTTP request object that defines the endpoint and method (e.g., GET, POST).
Request	`req_perform()`	Sends the built request to the server and *returns the response object.*
Response	`resp_body_json()`	Parses the response body as JSON and returns it as an R list.
Response	`resp_status()`	Retrieves the HTTP status from the response object.

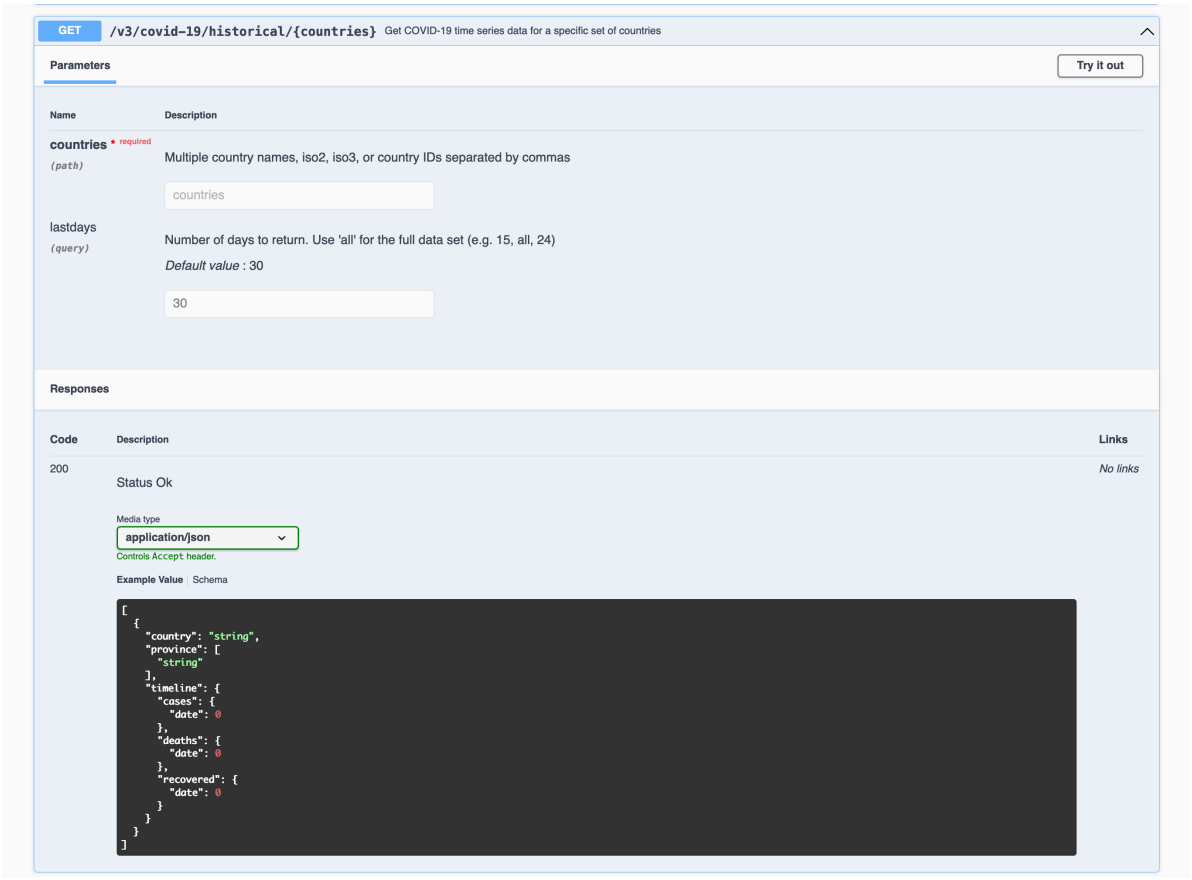
Build your own function wrapping the COVID API

Let's take stock of the datasets that we've gathered so far: we have swiss gdp and energy consumption. Now, let's say we want to look at how the COVID pandemic has affected the Swiss Economy. To understand this better, we would need to look at COVID case - and vaccination rates.

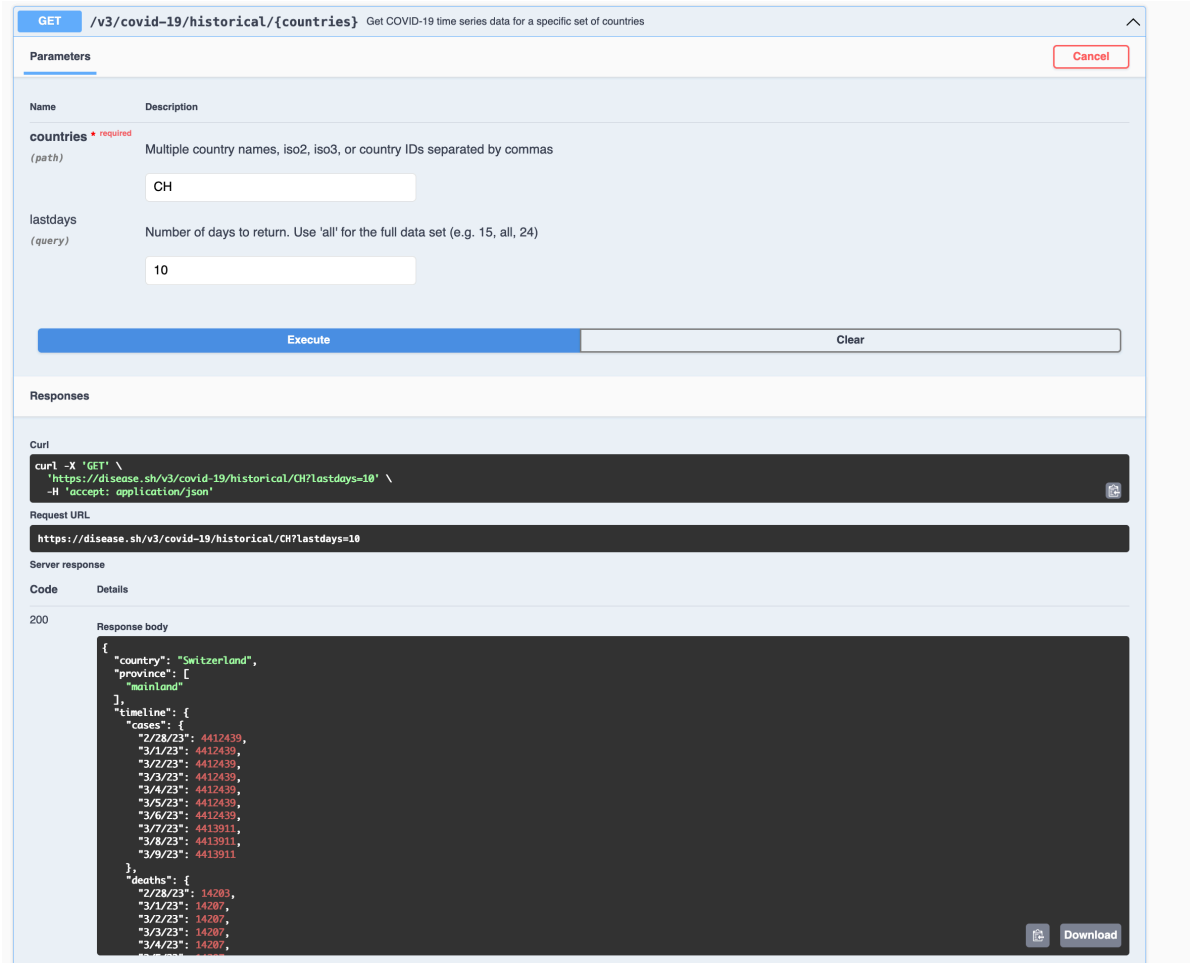
Luckily, the disease.sh docs which provide not only an open API for disease-related statistics, but also nice API documentation, so we can look at various APIs and try out their endpoints! Try it out!



Here we can see various COVID-19 APIs, such as from Worldometer or Johns Hopkins University.



Selecting one endpoint to try out, we can identify the historical case counts of certain countries.



When trying out the endpoint, we can even get the data as responses, and see all of the components of a real API call.

Let's try to turn this example into an actual function

```r
library(httr2)

get_case_counts <- function(country = "CH", period = c(30,365, "all")){
  # "https://disease.sh/v3/covid-19/historical/CH?lastdays=30"
  # this way is error prone, try to match.args to check if the inputs are correct
  base_url <- "https://disease.sh/v3/covid-19/historical"
  final_url <- paste0(base_url, "/", country, "?lastdays=", period)

  # perform API call with httr2
  req <- request(final_url)
```

```
  resp <- req_perform(req)

  # if request is not successful
  if (resp_status(resp) != 200){
    message("The request was not successful")
  }
  else{
    return(resp_body_json(resp))
  }
}

get_case_counts("CH", 1)
```

When running the last line `get_case_counts("CH", 1)` we get the following output

```
$country
[1] "Switzerland"

$province
$province[[1]]
[1] "mainland"


$timeline
$timeline$cases
$timeline$cases$`3/9/23`
[1] 4413911


$timeline$deaths
$timeline$deaths$`3/9/23`
[1] 14210


$timeline$recovered
$timeline$recovered$`3/9/23`
[1] 0
```

Here it's important to note that the Johns Hopkins University apparently stopped releasing this data after 3.9.2023 and it does seem a bit implausible that there are 0 recoveries on that day - alas!

Your turn!

Use the same documentation as above (disease.sh docs) to get the historical vaccination rates of a country.

Try out the desired endpoint using the documentation, before writing your own function.

Feel free to use this function as a template:

```
library(httr2)

get_country_vaccine_rates <- function(country = "CH", period = c(30,365, "all")){

  base_url <- "..."
  final_url <- "..."

  # perform  API call with httr2
  req <- "..."(final_url)
  resp <- "..."(req)

  # if request is not successful
  if (resp_status(resp) != 200){
    message("The request was not successful")
  }
  else{
    return(resp_body_json(resp))
  }
}
```

Try out your built function! What does it return?

Congratulations on adding one more dataset to your collection for your economic indicator.

But, it doesn't stop here! There are many other API (Wrappers) in R that you can use for your future analyses, to name a few:

- the kofdata R Package (for KOF data)
- the BFS R package (for Swiss federal statistics)
- the fredapi Python Library (for US economic data)
- the fredr data from the Federal Reserve Economic Data (FRED) API R package
- public-apis a Github repository listing Public APIs

In case you have any questions, feel free to contact me :) heim@kof.ethz.ch