API

Day 8

Solveig Bjørkholt

6. July

Plan for today

What we will learn today:

- What is an API?
- How to make an API request in R
- API formats
- Nested dataframes
- R-packages for API requests

What is an API?

Have you ever talked to somebody about data collection and then gotten the question "do they have an API?". For somebody who has never used APIs before, the concept can seem rather foreign, but it is extremely useful to know what APIs are and how to use them. API stands for **Application Programming Interface**, and basically, it is a way for the computer to handle contents on a webpage so that the computer can receive and deliver data easily. In other words, APIs are built for data transfer. For us, APIs improve upon previous ways of gathering data:

1. Sending a mail and asking for some flat files (e.g. csv), often time-consuming and not always feasible.
2. Applying for access to others' databases, which is often quite dubious.
3. Webscraping content, meaning writing long scripts and adding pressure to the client's server.

To be fair, APIs are about more than gathering data for analysis. It also allows webpages to communicate with each other, and with you. When you for example order tickets from a travelling agency, it's typically an API in the middle that tells you which days are available for travel, and that saves your requests. However, for our purposes, the *data gathering* part is the important one.

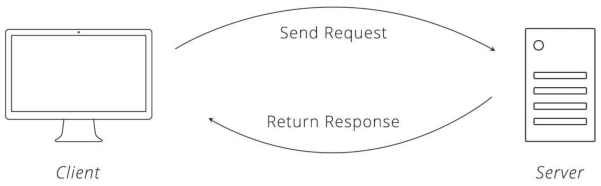
How to make an API request?

API requests consist of four things:

1. **Endpoint** – A part of the URL you visit. For example, the endpoint of the URL https://example.com/predict is /predict

2. **Method** – A type of request you're sending, can be either GET, POST, PUT, PATCH, and DELETE. They are used to perform one of these actions: Create, Read, Update, Delete (CRUD). Since we are mostly collecting data, GET is the most relevant method for us.
3. **Headers** – Used for providing information (think authentication credentials, for example). They are provided as key-value pairs.
4. **Body** – Information that is sent to the server. Used only when not making GET requests, and thus not very relevant for us.

The most important part for us, is to define the *endpoint*. This is where we tell the computer which data we want. We'll be using GET as a *method* to collect the data. The webpage we will be using for this example provides open data, so we do not need to authenticate ourselves via the *headers*. Using GET requests also makes the *body* argument obsolete, because we are not going to provide data to the server (as we for example do when we place an order in a travel agency).

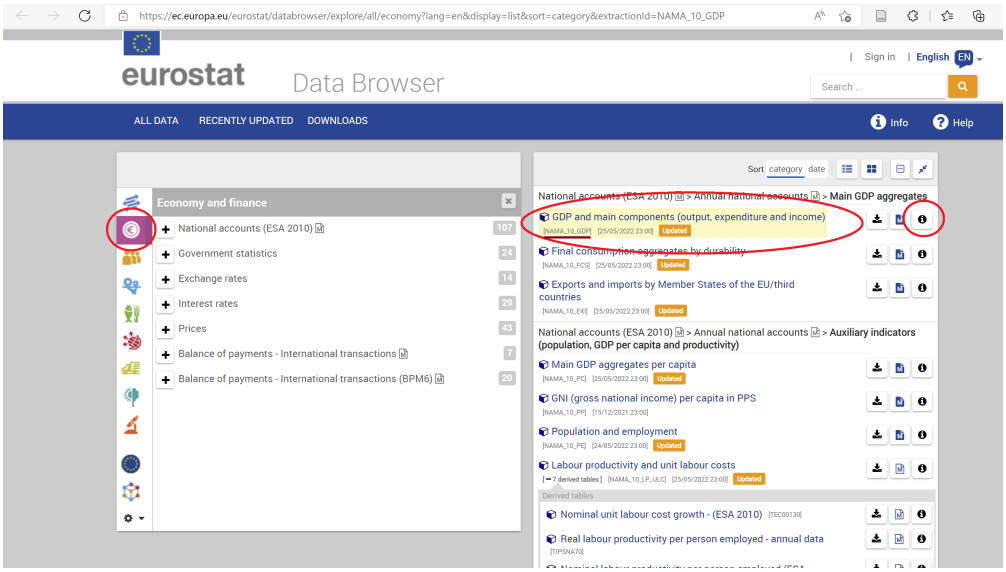


Eurostat example

Building the API request

To illustrate how making an API request works in practice, let's look at data provided by Eurostat. Eurostat is the organization for official statistics in Europe. They write about their API here. It's a method they provide to extract data from their servers. These data are also available from their their old-fashioned database, and admittedly, if you have never worked with Eurostat-data before, it is usually better to start here to get an overview of the data.

Now, when we want to extract data, we need to be aware of a few things. First and foremost we want to know which dataset we want. Let's assume we're looking for data on GDP in European countries. Searching the Eurostat database, we find the *GDP and main components (output, expenditure and income)* dataset. You can find the dataset here. Try opening the link and clicking on the information symbol to the right of the dataset. Here we see that the "online data code" is "NAMA_10_GDP". This is the name of the dataset containing information on GDP in European countries.



Let's try to fetch the GDP dataset using the API. Recall that an API request consists of an *endpoint*, and this is what we define in order to get what we want from the server. The endpoint is placed at the end of the host URL. Eurostat explains on their webpage that the fixed part (the host URL) for their API is

http://ec.europa.eu/eurostat/wdds/rest/data/v2.1/json/en/[1].

This is the fixed part of the URL, and we add the information we need to the end of this URL – thus making the *endpoint*.

What information do we need to add? The other things we need to know when making an API request to Eurostat, is:

1. Which variables we need (value added, individual consumption, GDP, etc.): NA_ITEM
2. Which measure on the variable we'd like (dollars, percentages, etc.): UNIT
3. Which units of observation (i.e. the countries or regions) we are interested in: GEO
4. Which years we need data for: TIME
5. Which time frequency (i.e. annual, quarterly, monthly, etc.) we want: FREQ

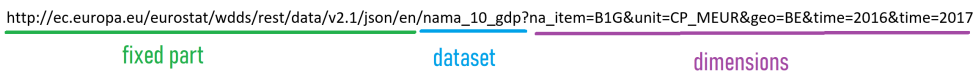
Let's call these "dimensions". Scrolling down in the information-window that we opened to find the name of the dataset, we see which choices we have on each dimension. For example, on the GEO-dimension we can choose regions such as European Union-27 countries (EU27_2020) or individual countries such as Belgium (BE). On the TIME dimension, we have years from 1975 to 2021. On the FREQ dimension, we have no other choices than annual data.

Let's say we want the dataset on GDP with:

1. **Value added, gross** as the variable (NA_ITEM)
2. **Current prices, million euro** as the measure (UNIT)
3. For the country **Belgium** (GEO)
4. For the years **2016** and **2017** (TIME)

Now we have a pretty specific wish, and this can be implemented into an API request. We add these dimensions to the endpoint, and the request will look like in the figure below. As you can see, this actually just one long URL! If you try typing it into your browser and hit enter, it wil display the dataset there (in JSON format).

[1]They are creating a new API which uses another host URL, but it is not fully operative yet.



Notice how we, through the URL, specify the **dataset**, the **na_item**, the **unit**, the **geo** and the **time**. To understand more of how this works, you can use the Eurostat query builder.

Get the data into R

To get these data into R using the Eurostat API, we have to load the package `httr`, a package used to handle API requests. Then, we make an object of the URL request we just made.

```
library(httr)

url <- "http://ec.europa.eu/eurostat/wdds/rest/data/v2.1/json/en/nama_10_gdp?na_item=B1G&unit=CP_MEUR&g
```

Second, we use the function `GET` from the package `httr` to make a GET request to the URL.

```
getresponse <- GET(url)

getresponse
```

```
## Response [http://ec.europa.eu/eurostat/wdds/rest/data/v2.1/json/en/nama_10_gdp?na_item=B1G&unit=CP_M
##   Date: 2022-06-21 12:13
##   Status: 200
##   Content-Type: application/json;charset=UTF-8
##   Size: 854 B
```

The GET request contains various information on whether the call was successful, how long it took to make it, when it was made, and so on. This information might be useful, for example the status `200` means that the call was successful. A status of `400` (including `401`, `402`, `403`, etc.) means that something went wrong, and the call was unsuccessful. You can read more about status codes here.

If the call was successful and everything is in order, we would ideally want the data. To extract the data from the object, we use the function `content`. Here, I specify the argument `as` to `"text"` to make a JSON-string out of the information. For illustrative purposes on what JSON actually looks like, I display the first 100 lines of the JSON-dataset using `substr`.

```
json <- content(getresponse, as = "text")

substr(json, 1, 100)
```

```
## [1] "{\"version\":\"2.0\",\"label\":\"GDP and main components (output, expenditure and income)\",\"h
```

Now that we have a JSON-object, we have all the information we need. But in R, we usually prefer dataframes to JSON. To work with JSON objects, we load a package that allows us to work with JSON, for example `rjstat`. From this package, we use the function `fromJSONstat` to make the JSON-file into a dataframe.

```
library(rjstat)

df <- fromJSONstat(json)

head(df)
```

```
##                            unit            na_item     geo time    value
## 1 Current prices, million euro Value added, gross Belgium 2016 384032.7
## 2 Current prices, million euro Value added, gross Belgium 2017 397034.3
```

We now have a dataframe with five variables and two rows. And we know the gross value added for Belgium in 2016 and 2017, measured in current prices (million euro).

To get a grasp of how APIs work, you can play around the the Eurostat query builder and use the code below to look at the datasets being produced. If you are unsure what certain dimensions mean, remember that you can always look them up in the Eurostat database.

```
library(httr)
library(rjstat)

url <- ""

getresponse <- GET(url)

json <- content(getresponse, "text")

df <- fromJSONstat(json)
```

API formats

APIs often give us semi-structured data. The formats are typically either JSON or XML, though you do see some APIs providing data in CSV-format as well. Let's take a brief look at how to interpret these data formats.

Let's say we have the dataset that we exctacted above – the dataset on Belgium's gross value added measured in million euro, in 2016 and 2017. For a standard (structured) dataframe, this dataset looks like this:

unit	na_item	geo	time	value
Current prices, million euro	Value added, gross	Belgium	2016	384032.7
Current prices, million euro	Value added, gross	Belgium	2017	397034.3

It's a dataframe with five variables and two rows.

In JSON, the structure of this dataset would be quite different. JSON works with so-called "key-value" pairs, which you can think of like specifying the group, and then specifying the variable in this group. They come in curly brackets {}, so you have a structure like:

{"key": "value"}

For example:

{"name": "Hannah"} {"city": "Bergen"}

In the square brackets [] you usually have the values of the variable. The hierarchical structure is given by indents and brackets.

```
[
   {
      "version":[
         "2.0"
```

```
    ],
    "label":[
       "GDP and main components (output, expenditure and income)"
    ],
    "href":[
       "http://ec.europa.eu/eurostat/wdds/rest/data/v2.1/json/en/nama_10_gdp?na_item=B1G&unit=CP_MEUR&
    ],
    "source":[
       "Eurostat"
    ],
    "updated":[
       "2022-05-25"
    ],
    "extension":{
       "datasetId":[
          "nama_10_gdp"
       ],
       "lang":[
          "EN"
       ],
       "description":{

       },
       "subTitle":{

       }
    },
    "class":[
       "dataset"
    ],
    "value":{
       "0":[
          384032.7
       ],
       "1":[
          397034.3
       ]
    },
    "dimension":{
       "unit":{
          "label":[
             "unit"
          ],
          "category":{
             "index":{
                "CP_MEUR":[
                   0
                ]
             },
             "label":{
                "CP_MEUR":[
                   "Current prices, million euro"
                ]
```

```
            }
         }
      },
      "na_item":{
         "label":[
            "na_item"
         ],
         "category":{
            "index":{
               "B1G":[
                  0
               ]
            },
            "label":{
               "B1G":[
                  "Value added, gross"
               ]
            }
         }
      },
      "geo":{
         "label":[
            "geo"
         ],
         "category":{
            "index":{
               "BE":[
                  0
               ]
            },
            "label":{
               "BE":[
                  "Belgium"
               ]
            }
         }
      },
      "time":{
         "label":[
            "time"
         ],
         "category":{
            "index":{
               "2016":[
                  0
               ],
               "2017":[
                  1
               ]
            },
            "label":{
               "2016":[
                  "2016"
```

```
                ],
                "2017":[
                   "2017"
                ]
             }
          }
       }
    },
    "id":[
       "unit",
       "na_item",
       "geo",
       "time"
    ],
    "size":[
       1,
       1,
       1,
       2
    ]
 }
]
```


XML-formats use tags to store information. As you can see, it's quite similar to HTML (which we looked at yesterday). The variable group is defined by the less-and-greater-than signs `<` and `>`, and the end of a tag has a slash before the name of the tag. The hierarchical structure is given by the number of indents. To work with XML-formats in R, you can use the package `xml`. The functions `xmlParse` and `xmlToList` should give you dataframes from the XML.

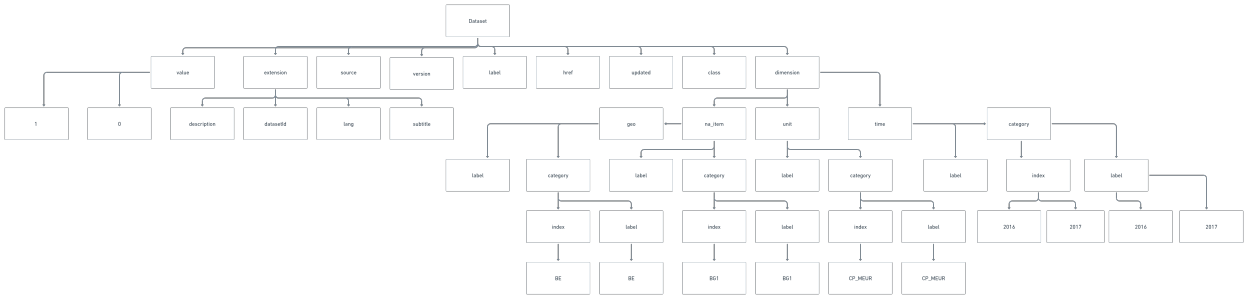

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <row>
    <version>2.0</version>
    <label>GDP and main components (output, expenditure and income)</label>
    <href>http://ec.europa.eu/eurostat/wdds/rest/data/v2.1/json/en/nama_10_gdp?na_item=B1G&amp;unit=CP_
    <source>Eurostat</source>
    <updated>2022-05-25</updated>
    <extension>
      <datasetId>nama_10_gdp</datasetId>
      <lang>EN</lang>
      <description/>
      <subTitle/>
    </extension>
    <class>dataset</class>
    <value>
      <0>384032.7</0>
      <1>397034.3</1>
    </value>
    <dimension>
      <unit>
        <label>unit</label>
        <category>
          <index>
```

```
            <CP_MEUR>0</CP_MEUR>
          </index>
          <label>
            <CP_MEUR>Current prices, million euro</CP_MEUR>
          </label>
        </category>
      </unit>
      <na_item>
        <label>na_item</label>
        <category>
          <index>
            <B1G>0</B1G>
          </index>
          <label>
            <B1G>Value added, gross</B1G>
          </label>
        </category>
      </na_item>
      <geo>
        <label>geo</label>
        <category>
          <index>
            <BE>0</BE>
          </index>
          <label>
            <BE>Belgium</BE>
          </label>
        </category>
      </geo>
      <time>
        <label>time</label>
        <category>
          <index>
            <2016>0</2016>
            <2017>1</2017>
          </index>
          <label>
            <2016>2016</2016>
            <2017>2017</2017>
          </label>
        </category>
      </time>
    </dimension>
    <id>unit</id>
    <id>na_item</id>
    <id>geo</id>
    <id>time</id>
    <size>1</size>
    <size>1</size>
    <size>1</size>
    <size>2</size>
  </row>
</root>
```

And if you have a CSV, you can for example use `read_csv` to get it into R.

Nested dataframes

The JSON and XML formats contain some extra information – a bit of metadata on the dataset. They tell us the version of the dataset, the name of the dataset, the source, the date it was updated, and so on. And, *nested* into this information is the information on the dimensions. For something to be nested means that it basically becomes and undercategory of something else. You can think of it as a tree structure. The nested structure in the JSON and XML above looks something like this:



Sometimes, we'd like to keep the nested format in R. For example, maybe you have a category "countries". Under that category you have categories for "Belgium", "France", "Germany" and so forth, and below that, data for GDP, population size and so forth. It is possible to keep the nested structure in R by creating nested dataframes. You can create your own nested dataframes by using the function `nest`, and then you can make them back to normal dataframes by using the function `unnest`. Below is an example of how you can read the full nested structure of a JSON-file into an R nested dataframe, and then unnest it.

```
library(jsonlite)

nested_df <- fromJSON("../../datafolder/jsonexample.json", flatten = TRUE)

nested_df %>%
  unnest()
```

```
## # A tibble: 4 x 26
##   version label          href  source updated class id     size extension.datas~
##   <chr>   <chr>          <chr> <chr>  <chr>   <chr> <chr> <int> <chr>
## 1 2.0     GDP and main ~ http~ Euros~ 2022-0~ data~ unit      1 nama_10_gdp
## 2 2.0     GDP and main ~ http~ Euros~ 2022-0~ data~ na_i~     1 nama_10_gdp
## 3 2.0     GDP and main ~ http~ Euros~ 2022-0~ data~ geo       1 nama_10_gdp
## 4 2.0     GDP and main ~ http~ Euros~ 2022-0~ data~ time      2 nama_10_gdp
## # ... with 17 more variables: extension.lang <chr>, value.0 <dbl>,
## #   value.1 <dbl>, dimension.unit.label <chr>,
## #   dimension.unit.category.index.CP_MEUR <int>,
## #   dimension.unit.category.label.CP_MEUR <chr>, dimension.na_item.label <chr>,
## #   dimension.na_item.category.index.B1G <int>,
## #   dimension.na_item.category.label.B1G <chr>, dimension.geo.label <chr>,
## #   dimension.geo.category.index.BE <int>, ...
```

R-packages for API requests

If you thought gathering data through the Eurostat API was cumbersome, you are not the first one to think that. Using APIs through the `httr` package is alright, but it can be quite tricky to get into sometimes. That is why, luckily, many package builders in the R community have created their own packages to use APIs easily in R. A quick search on the internet tells us that someone have created an R package for the Eurostat API as well!

The package is called eurostat. Remember, the first time you use a package you have to `install.packages()`. After that, we can load it into R.

```
library(eurostat)
```

```
## Warning: package 'eurostat' was built under R version 4.1.3
```

The package has many useful functions. `get_eurostat_toc`, for example, gives us a table of contents for the eurostat databases. Here, we can browse to find datasets useful for our analysis.

```
get_eurostat_toc() %>%
  head() # Displays the first six rows of the dataset
```

```
## # A tibble: 6 x 8
##   title    code  type  `last update o~` `last table st~` `data start` `data end`
##   <chr>    <chr> <chr> <chr>            <chr>            <chr>        <chr>
## 1 Databas~ data  fold~ <NA>             <NA>             <NA>         <NA>
## 2 General~ gene~ fold~ <NA>             <NA>             <NA>         <NA>
## 3 Europea~ euro~ fold~ <NA>             <NA>             <NA>         <NA>
## 4 Busines~ ei_b~ fold~ <NA>             <NA>             <NA>         <NA>
## 5 Consume~ ei_b~ fold~ <NA>             <NA>             <NA>         <NA>
## 6 Consume~ ei_b~ data~ 30.05.2022       30.05.2022       1980M01      2022M05
## # ... with 1 more variable: values <chr>
```

With the function `search_eurostat`, we can search for datasets for specific keywords. Using this function, we see that our dataset nama_10_gdp comes high up on the list.

```
search_eurostat("GDP") %>%
  head()
```

```
## # A tibble: 6 x 8
##   title    code  type  `last update o~` `last table st~` `data start` `data end`
##   <chr>    <chr> <chr> <chr>            <chr>            <chr>        <chr>
## 1 GDP and~ nama~ data~ 21.06.2022       22.03.2022       1975         2021
## 2 GDP and~ namq~ data~ 16.06.2022       29.04.2022       1975Q1       2022Q1
## 3 Gross d~ nama~ data~ 18.04.2022       18.04.2022       2000         2020
## 4 Average~ nama~ data~ 18.04.2022       18.04.2022       2000         2020
## 5 Gross d~ nama~ data~ 18.04.2022       18.04.2022       2000         2020
## 6 Europea~ ipr_~ data~ 20.12.2016       08.02.2021       2000         2014
## # ... with 1 more variable: values <chr>
```

To get the data into R, we use the function `get_eurostat`.

```
get_eurostat("nama_10_gdp") %>%
  head()
```

```
## Table nama_10_gdp cached at C:\Users\solvebjo\AppData\Local\Temp\RtmpqUYqH2/eurostat/nama_10_gdp_dat
```

```
## # A tibble: 6 x 5
##   unit       na_item geo   time        values
##   <chr>      <chr>   <chr> <date>       <dbl>
## 1 CLV05_MEUR B1G     AT    2021-01-01 270365.
## 2 CLV05_MEUR B1G     BA    2021-01-01  11154
## 3 CLV05_MEUR B1G     BE    2021-01-01 343926.
## 4 CLV05_MEUR B1G     BG    2021-01-01  29422.
## 5 CLV05_MEUR B1G     CH    2021-01-01 440225.
## 6 CLV05_MEUR B1G     CY    2021-01-01  17298
```

And indeed, you can filter in the R-code just like we did above in the URL by adding the argument `filter` and wrap the filters into a list. To learn more about how to use the Eurostat API R-package, take a look at this link.

```
get_eurostat("nama_10_gdp",
             filters = list(na_item = "B1G",
                            unit = "CP_MEUR",
                            geo = "BE",
                            time = c("2016", "2017"))) %>%
  head()
```

```
## # A tibble: 2 x 5
##   unit    na_item geo   time        values
##   <chr>   <chr>   <chr> <date>       <dbl>
## 1 CP_MEUR B1G     BE    2016-01-01 384033.
## 2 CP_MEUR B1G     BE    2017-01-01 397034.
```