Data Science and Visualization, S2026

Interacting with APIs

Jonas Smedegaard [1]

[1]Roskilde University, Department of People and Technology

2026-03-10

Exercise 1: Identifying API endpoints

In this exercise we will use the World Bank Data360 API to retrieve data on the following 10 World Development Indicators (WDI) for the time period 1990-2023 for member states of the European Union as of 2026:

- Population, total
- Population ages 65 and above (% of total population)
- Life expectancy at birth, total (years)
- GDP (Current US$)
- GDP growth (annual %)
- GDP per capita (Current US$)
- Inflation, consumer prices (annual %)
- Unemployed, total (% of labor force)
- Educational attainment, at least Bachelor's or equivalent, population 25+, total (%) (cumulative)
- Gini index

1.1 Use the empty code cell below to first install the Requests library. Then clear the cell and use it to import the pandas, Plotnine, and Requests libraries

```
# Import libraries
import requests as rq
```

1.2 Use the WDI documentation to find the identifiers for the WDIs listed above. You can search for the relevant WDIs in the WDI documentation. Navigate to the webpage for the relevant WDI, then use either the URL or click the button *details* to find the identifier. When you have the identifiers, use the empty code cell below to to construct the API endpoints that will retrieve the data described above. The syntax for endpoints is described in the Data360 API documentation. Store the API endpoints in a list. Here is some information to get you started:

- The WDI documentation uses periods whereas the Data360 API uses underscores as separators in the identifiers. The Data360 API also expects the suffix *WB_WDI* for each identifier
- The Data360 API returns a maximum of 1000 records per call. For this reason you will have to construct separate API endpoints for each WDI
- The Data360 API uses ISO-3 codes to identify countries. You can find a list of member states of the European Union as of 2026 on this Wikipedia page and a list of ISO-3 codes on this Wikipedia page

```
# SPDX-FileCopyrightText: 2026 Jonas Smedegaard <dr@jones.dk
# SPDX-License-Identifier: GPL-3.0-or-later
import urllib.request
import urllib.error
import requests_cache

cache_session = requests_cache.CachedSession(
    'sparql2pandas',
    backend='sqlite',
    compression='gzip',
    allowable_methods=('POST'),
)

def cached_urlopen(request, timeout=None):
    response = cache_session.post(
        request.get_full_url(),
        data=request.data,
        headers=dict(request.header_items()),
        timeout=timeout or 30,
    )
    if response.status_code >= 400:
        raise urllib.error.URLError(f"HTTP {response.status_code}")
    return type('CachedResponse', (), {
```

```
        'read': lambda self: response.content,
        'geturl': lambda self: response.url,
        'info': lambda self: response.headers,
        '__enter__': lambda self: self,
        '__exit__': lambda *args: None,
    })()

urllib.request.urlopen = cached_urlopen

from SPARQLWrapper.SmartWrapper import SPARQLWrapper2, Bindings
import pandas as pd
import time

def _topandas(self) -> pd.DataFrame:
    """Convert SPARQL Bindings to DataFrame."""
    return pd.DataFrame(
        [[binding.get(var).value if binding.get(var) else None
          for var in self.variables]
         for binding in self.bindings],
        columns=self.variables)

Bindings.topandas = _topandas

def sparql2pandas(endpoint, rate_limit, query):
    time.sleep(rate_limit)
    sparql = SPARQLWrapper2(endpoint)
    sparql.agent = "SPARQL2Pandas"
    sparql.setMethod("POST")
    sparql.setQuery(query)
    return sparql.query().topandas()
```

```
query = """
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX p: <http://www.wikidata.org/prop/>
PREFIX ps: <http://www.wikidata.org/prop/statement/>
PREFIX pq: <http://www.wikidata.org/prop/qualifier/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

```

```
SELECT ?countryLabel ?iso3Code WHERE {
  # Define targetDate (hint: Brexit occured at 2020-01-31T23:00:00Z)
  BIND("2026-01-01T00:00:00Z"^^xsd:dateTime AS ?targetDate)

  # Find statements about EU membership (not just the direct property)
  ?country p:P463 ?statement .
  ?statement ps:P463 wd:Q458 .

  # Check that membership started on or before the target date
  ?statement pq:P580 ?startTime .
  FILTER(?startTime <= ?targetDate)

  # Check that membership either has no end time (still active)
  # OR ended after the target date
  OPTIONAL { ?statement pq:P582 ?endTime }
  FILTER(!BOUND(?endTime) || ?endTime > ?targetDate)

  # Get the ISO 3166-1 alpha-3 code
  ?country wdt:P298 ?iso3Code .

  # Get the label (name) of the country
  #SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
  ?country @en@rdfs:label ?countryLabel .
  #?country rdfs:label ?countryLabel. FILTER(LANG(?countryLabel) = "en")
}
ORDER BY ?countryLabel

"""

#df = sparql2pandas("https://query.wikidata.org/sparql", 60, query)
df = sparql2pandas("https://qlever.dev/api/wikidata", 0, query)
print(df)
```

```
      countryLabel iso3Code
0          Austria      AUT
1          Belgium      BEL
2         Bulgaria      BGR
3          Croatia      HRV
4           Cyprus      CYP
5   Czech Republic      CZE
```

```
6          Denmark      DNK
7          Estonia      EST
8          Finland      FIN
9           France      FRA
10         Germany      DEU
11          Greece      GRC
12         Hungary      HUN
13         Ireland      IRL
14           Italy      ITA
15          Latvia      LVA
16       Lithuania      LTU
17      Luxembourg      LUX
18           Malta      MLT
19          Poland      POL
20        Portugal      PRT
21         Romania      ROU
22        Slovakia      SVK
23        Slovenia      SVN
24           Spain      ESP
25          Sweden      SWE
```

```
## Construct API endpoints
# Start by assigning base URL and service components to string variables
base_URL = "https://api.worldbank.org/v2/"
service = "country/aut;bel;bgr;cyp;cze;deu;dnk;esp;est;fin;fra;grc"\
+ ";hrv;hun;irl;ita;ltu;lux;lva;mlt;nld;pol;prt;rou;svk;svn;swe"\
+ "/indicator/"
scope = "?date=1990:2023&format=json&per_page=20000"

# Create a list of WDI identifiers

## Population, total
pop_totl = "SP.POP.TOTL"
## Population ages 65 and above (% of total population)
pop_65up = "SP.POP.65UP.TO.ZS"
## Life expectancy at birth, total (years)
life_expect = "SP.DYN.LE00.IN"
## GDP (Current US\$)
gdp = "NY.GDP.MKTP.CD"
## GDP growth (annual %)
gdp_growth = "NY.GDP.MKTP.KD.ZG"
```

```
## GDP per capita (Current US$)
gdp_per_capita = "NY.GDP.PCAP.CD"
## Inflation, consumer prices (annual %)
inflation = "FP.CPI.TOTL.ZG"
## Unemployed, total (% of labor force)
unemployed = "SL.UEM.TOTL.ZS"
## Educational attainment, at least Bachelor's or equivalent,
## population 25+, total (%) (cumulative)
edu = "SE.TER.CUAT.BA.ZS"
## Gini index
gini = "SI.POV.GINI"

# Create a string variable
# containing ISO-3 codes for EU member countries
eu = "aut;bel;bgr;cyp;cze;deu;dnk;esp;est;fin;fra;grc;hrv;hun;irl"\
+ ";ita;ltu;lux;lva;mlt;nld;pol;prt;rou;svk;svn;swe"

# Make a list of queries

# Make a list of endpoints
```

Exercise 2: Making GET requests

In the empty code cell below, use the function `rq.get()` and the endpoints you constructed above to make GET requests to the Data360 API. As you will need to make a separate GET request for each WDI, a good approach is to use a `for` loop to make the requests. Store the responses in a list. When you have made the requests, print the HTTP status code for each response

```
# Make GET requests to the Data360 API

# Print the HTTP status codes for each response
```

Exercise 3: Parsing API responses

3.1 Create a dictionary that contains the parsed JSON responses from the API. Use the `.json()` method to parse the contents of the responses. When you have parsed the responses and created the dictionary, examine the structure of the first parsed response

```
# Create a dictionary of parsed responses

# Examine the structure of the first parsed response
```

3.2 To work with the data we need to convert it to a DataFrame where each rows contains the values of the WDIs for a given member state of the European Union in a given year between 2000 and 2020. This type of tabular data structure is what economists call panel data. There are several ways to transform the API responses to a panel data set. One approach is to follow these 5 steps:

1. Start by creating an empty nested dictionary. Let the keys of the outer dictionary be the WDI identifiers you found in exercise 1.1. Let the keys of the inner dictionaries be "ISO-3", "year", and the WDI identifier corresponding to the outer dictionary key. Let the values of the inner dictionaries be empty lists. You can create the empty nested dictionary with a `for` loop
2. Use a nested `for` loop to go through the inner dictionaries of the parsed API responses and populate the empty nested dictionary you just created with ISO-3 codes, years, and values of the WDIs. The inner dictionary keys in the parsed API responses that you will need are 'REF_AREA', 'TIME_PERIOD', and 'OBS_VALUE'
3. Then use the nested dictionary you just populated to create a list of DataFrames. As usual, we can create a new DataFrame from a dictionary of lists with the function `pd.DataFrame`
4. Now use a `for` loop to iteratively merge the DataFrames contained in the list you just created to one large DataFrame containing all the data. Start by makeing an empty DataFrame with columns "ISO-3" and "year". You can then merge on the columns "ISO-3" and "year". Make sure to use outer joins to no drop in any values. Name the large DataFrame `df`
5. Finally, rename the WDI columns of the large DataFrame to something shorter and meaningful. You can use the function `pd.rename()` for this. Then output the DataFrame to see how it looks

```
# Step 1: Create an empty nested dictionary

# Step 2: Use a nested loop to through the response dictionaries
# and populate the empty dictionary

# Step 3: Make a list of dataframes based on the populated dictionary

# Step 4: Merge the DataFrames in the list into one large DataFrame

# Step 5: Rename the columns of the large DataFrame

# Output df
```

Exercise 4: Detecting missing values

Use the empty code cells below to create a stacked bar plot that shows the proportion of missing and non-missing values on each column in `df`. You can do this by following these 3 steps:

1. Start by creating a new DataFrame where the first column contains all the column names of `df` and the second and third columns contain the percentage of missing and non-missing values in a given column. Than reshape the new DataFrame such that there are now two rows oer column of `df` and the percentages of missing and non-missing values are contained in a single column. A third column should identify whether the percentage value in a given row is for missing or non-missing values. Do this using the .melt() method
2. Order column names by percentage of missing values and remove percentages equal to zero to increase readability of the plot
3. Create the plot with Plotnine. Use the function `geom_col()`, display column names on the vertical axis, the percentages on the horizontal axis, and map the identifyer column to the `fill` aesthetic. Select colors to use for missing and non-missing values with the function `scale_fill_manual()`. Label each slice of the bars with the percentage it represents

As always, make sure your plots abide to the principles of data visualization. One approach to controlling the layout of your plots is to specify a global theme with the `theme()` function. You can then modify the theme for each plot you create as needed. When you have created the plot, you will see that we have some column with a lot missing values. However, as there is not any good method for replacing the missing values immediately available to us, we will ignore the problem

```
# Specify a global theme to use in all plots
```

```
# Step 1: Construct a DataFrame containing the data to be plotted
```

```
# Step 2: Reorder categorical variables and replace 0 with none
```

```
# Step 3: Create the plot
```

```
# Output the plot
```

Exercise 5: Descriptive Statistics and Data Visualizations

5.1 Use the empty code cell below to create table displaying the number of observations, mean, median, standard deviation, minimum, and maximum of GDP per capita and GDP growth, and life expectancy at birth *separately for each member state of the European Union*. Use the pandas methods `groupby()` and `agg()` to do this. Order the table by the mean of GDP per capita such that the country with the highest mean GDP per capita for the period 1990-2023 appears in the first rows

```
# Create a table displaying summary statistics
# separately for each country
```

5.2 Use the empty code cell below to visualize the development of GDP per capita over the period 1990-2023 for (1) the European Union mean, (2) the two member states with the lowest mean GDP per capita over the period, and (3) the two member states with highest GDP per capite over the period. You can do this by following this two steps:

1. Create a DataFrame containing the time series to plot. You can combine groupwise aggregation with the pandas `.groupby()` method, indexing via the `loc` operator, merging using the `.merge()` method, and reshaping using the `.melt()` method to do this. Use the table that you

created in exercise 5.1 to identify the member states of the European Union with lowest and highest mean GDP per capita over the period. You will need to put all the GDP per capita values to be plotted in a single column
2. Create the plot. Use the functions `geom_point()` and `geom_line()` to plot the data. Assign the values to the vertical axis of the coordinate system and years to the horizontal axis. Use aesthetics to let lines and points represent the different time series contained in the plot

```
# Step 1: Create a DataFrame containing the data to be plotted
```

```
# Step 2: Create the plot
```

```
# Output the plot
```

5.3 Use the empty code cells below to visualize the development of mean GDP growth and the mean inflation rate over the period 2000-2023, separately for the member states of the European Union located in Central and Eastern, Western, Northern and Southern Europe, respectively. You can do this by following this two steps:

1. Start by creating a new column in `df` that groups member states of the European Union into Central and Eastern, Western, Northern, and Southern European Countries. Then create a DataFrame containing the time series to be plotted. You can combine groupwise aggregation with the `.groupby()` method and reshaping using the `.melt()` method to do this. You will need to construct a DataFrame where the first column contains year, the second column contains the geographic division, the third columns contains identifiers for the WDIs to be plotted, and the fourth column contains the values
2. Create the plot. Use the functions `geom_point()` and `geom_line()` to plot the data. Assign the values to the vertical axis of the coordinate system and years to the horizontal axis. Use aesthetics to let lines and points to represent the WDIs in the plots. Map the European regions to small multiples using the function `facet_wrap()`

```
# Step 1: Create a DataFrame containing the data to be plotted
```

```
# Step 2: Create the plot
```

```
# Output the plot
```

5.4 Create a scatterplot of the the fraction of the population with at least a bachelor's degree and life expectancy at birth. Group the data points by European regions. Include only values for the period 2010-2023 in the plot. You can create the plot by using the function `geom_point()`. Map the columns frac_higher_edu and LEB to the position scales of the coordinate system. Map the column containing European regions to the `fill` aesthetic

```
# Create the plot
```

```
# Output the plot
```