Data Science for Public Policy

Alex Engler & Aaron R. Williams - Georgetown University

Application Programming Interfaces (APIs)

Background

An *Application Programming Interface (API)* is a set of functions for programmatically accessing and/or processing data. A web API is a way to interact with web-based software through code - usually to retrieve data or use a tool. We will only focus on web APIs. Actions between computers and web applications are typically communicated through URLs that are passed between the computers.

Popular Web APIs

- Google Maps APIs
- Twitter API
- GitHub API
- World Bank
- Census API
- OpenFEC
- Urban Institute Education Data Portal
- Stanford CoreNLP API
- OpenTripPlanner API

Why APIs?

Why use APIs?

- **The data or method is only available through an API:** Census data and IPEDS are available through point-and-click interfaces and APIs, but many data and methods are only available through APIs. For example, data from the GoogleMaps API and methods from the Stanford CoreNLP software are only available in R through APIs.
- **APIs promote reproducibility:** Even with good documentation, pointing-and-clicking is tough to reproduce. APIs are useful for writing code to access data in an analysis before processing, modeling, and/or visualizing.

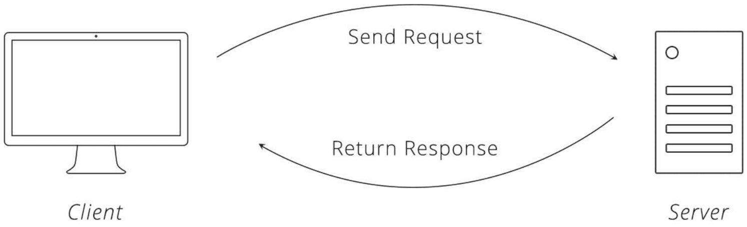
- **APIs enable scalability:** It's simple to point-and-click data for one state or one county. It's much tougher to point-and-click state or county level data if the data are stored in separate files. APIs can be accessed programtically, which means anything can be iterated to scale.

Why build APIs?

- **APIs enable scalability:** APIs enable scalability for data or methods creators. For example, the Urban Institute builds data visualization applications like Explore Your School's Changing Demographics on top of APIS.
- **APIs create standardization:** APIs standardize the communication between applications. By creating a protocol for a store of data or a model, it makes it simpler for downstream applications to access the data or methods of upstream applications.
- **Democratize access to data or methods:** Centralized storage and processing on the cloud are cheaper than personal computing on desktops and laptops. APIs allow users to access subsets of data or summary statistics quicker and cheaper than accessing all data and/or processing that data locally. For example, the new Urban Institute prototype summary API endpoints allow users to create summary statistics in API calls, which saves time and local storage.

Technical Background Part I

- **Server:** Remote computer that stores and processes information for an API. Note: This does not need to be a literal server. Many APIs are serverless and hosted in the cloud.
- **Client:** Computer requesting information from the API. In our cases, this is your computer.
- **Hyper-Text Transfer Protocol (HTTP):** Set of rules clients use to communicate with servers via the web (etiquette).
- **Request:** Contacting a web API with specifics about the information you wish to have returned. In general, this is built up as a URL.
- **Response:** A formatted set of information servers return to clients after requests.



Source: Zapier

- **Uniform Resource Locators (URL):** Text string that specifies a web location, a method for retrieving information from that web location, and additional parameters. We use these every day!

Process (without R code)

1. Client builds up a URL and sends a request to a server
2. Client receives a response from the server and checks for an error message
3. Client parses the response into a useful object: data frame

1. REQUEST (build up a URL)

Let's walk through an example from the Census Bureau.

1. The Census Bureau API contains hundreds of distinct data sets. Copy-and-paste https://api.census.gov/data.html into your web browser to see the list of datasets and documentation. Each one of these data sets is an API resource.
2. All Census Bureau API datasets are available from the same host name. Copy-and-paste https://api.census.gov/data/ into your browser. This contains information about all of the datasets in the previous link, only now the information is stored as JSON instead of HTML.
3. Each dataset is accessed through a *base URL* which consists of the host name, the year, and the "dataset name". Copy-and-paste https://api.census.gov/data/2014/pep/natstprc into your web browser. This returns Vintage 2014 Population Estimates: US, State, and PR Total Population and Components of Change. Each base URL corresponds to an API endpoint.
4. At this point, we're still only seeing metadata. We need to add a query string with a method and parameters. Copy-and-paste https://api.census.gov/data/2014/pep/natstprc?get=STNAME,POP&DATE_=7&for=state:* into your browser. Note the query string begins with `?`, includes the get method, and includes three parameters.

2. Check for a server error in the response

1. All of the URLs in part 1 were correctly specified and all of them returned results. What happens when incorrect URLs are passed to an API? Open a new browser window and copy-and-paste https://api.census.gov/data/2014/pep/natstprc?get=STNAME,POP&DATE_=7&for=state:57. Note: this call requests information for FIPs 57, which does not exist.

3. Parse the response

APIs need to return complicated hierarchical data as text. To do this, most APIs use **JavaScript Object Notation (JSON)**. JSON is a plain text hierarchical data structure. JSON is not JavaScript code. Instead, it's a non-rectangular method for storing data that can be accessed by most web applications and programming languages. Lists are made with `[]`. Objects are made with `{}` and contain key-value pairs. JSON is good at representing non-rectangular data is very much a web standard.

```
{
  "Class name": "Intro to Data Science",
  "Class ID": "PPOL 670",
  "Instructors": ["Aaron R. Williams", "Alex C. Engler"],
  "Location": {
    "Building": "Healy Hall"
    "Room": "105"
  }
}
```

Web APIs can also return Extensible Markup Language (XML) and HyperText Markup Language (HTML), but JSON is definitely most popular. We'll use `library(jsonlite)` to parse hierarchical data and turn it into tidy data.

Technical Background Part II

- **API Resource** An object in an API. In our case, a resource will almost always be a specific data set.
- **API endpoint** The point where a client communicates with a web API.
- **Method/verb:** A verb that specifies the action the client wants to perform on the resource through an API endpoint:
  - GET - Ask an API to send something to you
  - POST - send something to an API
  - HEAD, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH
- **Query string** Part of a URL that assigns values to parameters. Query strings begin with `?`. The general form is **key=value**.
- **Parameters:** Arguments that are passed to APIs. Parameters are separated by `&`.
- **Headers** Meta-information about GET requests. We will only use the User-Agent HTTP Header.
- **Body** Additional information sent to the server. This isn't import for GET requests.
- **Server response code** (HTTP status code)
  - 100s: information responses
  - 200s: success
  - 300s: redirection
  - 400s: client error
  - 500s: server error

R example 1

Let's walk through the example above using R. First install `library(httr)` and `library(jsonlite)` with `install.packages(c("httr", "jsonlite"))`.

`httr` contains tools for working with HTTP and URLs. It contains functions for all HTTP methods including `GET()` and `POST()`.

1. REQUEST (build up a URL)

Using the final URL from the above example, lets query state names and state population in July 2014 for all states, Washington, D.C., and Puerto Rico. Note, it is good practice to create the link outside of `GET()` because we may need to manipulate the URL string in later examples.

```
library(tidyverse)
library(httr)
library(jsonlite)

# make the URL
url <-
  "https://api.census.gov/data/2014/pep/natstprc?get=STNAME,POP&DATE_=7&for=state:*"

# use the URL to make a request from the API
pop_json <- GET(url = url)
```

2. Check for a server error in the response

```
http_status(pop_json)
```

```
## $category
## [1] "Success"
##
## $reason
## [1] "OK"
##
## $message
## [1] "Success: (200) OK"
```

3. Parse the response

```
# get the contents of the response as a text string
pop_json <- content(pop_json, as = "text")

# create a character matrix from the JSON
pop_matrix <- fromJSON(pop_json)

# turn the body of the character matrix into a tibble
pop_data <- as_tibble(pop_matrix[2:nrow(pop_matrix), ],
                      .name_repair = "minimal")

# add variable names to the tibble
names(pop_data) <- pop_matrix[1, ]

pop_data
```

```
## # A tibble: 52 x 4
##    STNAME               POP      DATE_ state
##    <chr>                <chr>    <chr> <chr>
##  1 Alabama              4849377  7     01
##  2 Alaska               736732   7     02
##  3 Arizona              6731484  7     04
##  4 Arkansas             2966369  7     05
##  5 California           38802500 7     06
##  6 Colorado             5355866  7     08
##  7 Connecticut          3596677  7     09
##  8 Delaware             935614   7     10
##  9 District of Columbia 658893   7     11
## 10 Florida              19893297 7     12
## # ... with 42 more rows
```

Terms of Service of User Agents

Always read an API's terms of service to ensure that use of the API conforms to the API's rules.

Furthermore, it is a good idea to only run one API request at a time and to identify yourself as a user-agent in the header of the HTTP request. This is simple with `user_agent()` from `library(httr)`:

```
GET(url_link,
    user_agent("Georgetown Univ. Student Data Collector (student@georgetown.edu)."))
```

Exercise 1

1. Visit the Urban Institute Education Data Portal API.

2. Read the Terms of Use.
3. Let's pull data from IPEDS about full-time equivalent enrollment in graduate schools in 2016 in Washington, DC (FIP = 11). Create a new .R script called get-ed-data.R.
4. Read the instructions for direct API access. Copy-and-paste the generic form of the API call into you .R script.
5. Navigate to the documentation for Enrollment - Full time equivalent. The example URL is "/api/v1/college-university/ipeds/enrollment-full-time-equivalent/{year}/{level_of_study}/" **Note:** The Urban Institute created an R package that simplifies this entire process. Let's build our request from scratch just to prove that we can.
6. Combine the endpoint URL with the example URL from earlier. This means combining the base URL with the URL and parameters for the endpoint.
7. Fill in the necessary parameters.
8. Make a GET request with `GET()`. Be sure to include the User-Agent.
9. Check the HTTP status.
10. Parse the response.

…

Authentication

Many APIs require authentication. For example, the Census API requires a user-specific API key for all API calls. (sign up now) This API key is simply passed as part of the path in the HTTP request. "https://api.census.gov/data/2014/pep/natstprc?get=STNAME,POP&DATE_=7&for=state:*&key=your key here" There are other authentication methods, but this is most common.

It is a bad idea to share API credentials. NEVER post a credential on GitHub. A convenient solution is to use a credentials file as follows:

```r
credential <- read_file("credential.txt")

url <- str_glue(
  "https://api.census.gov/data/2014/pep/natstprc?get=STNAME,POP&DATE_=7&for=state:*&key={credential}"
)
```

Be sure to add this credentials file to your `.gitignore`!

Pagination

A single API call could potentially return an unwieldy amount of information. This would be bad for the server, because the organization would need to pay for lots of resources. This would also be bad for the client because the client could quickly become overwhelmed by data. To solve this issue, many APIs are paginated. Pagination is simply breaking API responses into subsets.

For example, the original example returned information for all states in the United States. When information is requested at the Census tract level, instead of returning information for the entire United States, information can only be requested at the state level. Getting information for the entire United States will require iterating through each state.

Rate Limiting

Rate limiting is capping the number of requests by a client to an API in a given period of time. This is most relevant when results are paginated and iterating requests is necessary. It is also relevant when developing code to query an API–because a developer can burden the API with ultimately useless requests.

It is sometimes useful to add `Sys.sleep()` to R code, to pause the R code to give the API a break from requests between each request. Even 0.5 seconds can be the difference.

R example 2 (Advanced)

This example pulls information at the Census track level. Because of pagination, the example requires a custom function and iterates that function using `map_df()` from `library(purrr)`. It includes `Sys.sleep()` to pause the requests between each query.

The example pulls the estimated number of males (B01001_002E) and females (B01001_026E) in the 2018 5-year ACS for each Census tract in Alabama and Alaska.

Here are a few select columns for the 2018 5-year ACS from the Census API documentation page:

Vintage	Dataset Name	Dataset Type	Geography List	Variable List	Group List	Examples
2018	acs>acs5>profile	Aggregate	geographies	variables	groups	examples

1. Write a custom function

```
get_acs <- function(fips, credential) {

  url <- str_glue(
    "https://api.census.gov/data/2018/acs/acs5?get=B01001_002E,B01001_026E&for=tract:*&in=state:01&key={cre
  )

  # use the URL to make a request from the API
  acs_json <- GET(url = url)

  # get the contents of the response as a text string
  acs_json <- content(acs_json, as = "text")

  # create a character matrix from the JSON
  acs_matrix <- fromJSON(acs_json)

  # turn the body of the character matrix into a tibble
  acs_data <- as_tibble(acs_matrix[2:nrow(acs_matrix), ],
                        .name_repair = "minimal")

  # add variable names to the tibble
  names(acs_data) <- acs_matrix[1, ]

  return(acs_data)

  # pause for a second
  Sys.sleep(0.5)
}
```

2. Create a vector of FIPs

```
fips <- c("01", "02")
```

3. Iterate the functions

```
# load credentials
credential <- read_file(here::here("tutorials", "05_apis", "credential.txt"))

# iterate the function over Alabama and Alaska
map_df(fips, .f = ~get_acs(fips = .x, credential = credential))
```

```
## # A tibble: 2,362 x 5
##    B01001_002E B01001_026E state county tract
##    <chr>       <chr>       <chr> <chr>  <chr>
##  1 1230        1232        01    073    005701
##  2 2064        2929        01    073    010704
##  3 2782        3266        01    073    012908
##  4 1778        1850        01    073    005302
##  5 5728        5807        01    073    011107
##  6 1543        1798        01    073    013300
##  7 3015        2869        01    073    014406
##  8 974         1093        01    073    013801
##  9 1692        2347        01    023    956800
## 10 1706        1736        01    023    956900
## # ... with 2,352 more rows
```

R Packages

There are R packages that simplify interacting with many popular APIs. `library(censusapi)` (tutorial here) simplifies navigating Census documentation, checking the status code, building URLs for the Census API, and parsing JSON responses. This can save a lot of time and effort! The following code is one iteration of the advanced example from above!

```
library(censusapi)

getCensus(name = "acs/acs5",
          vars = c("B01001_002E", "B01001_026E"),
          region = "tract:*",
          regionin = "state:01",
          vintage = 2018,
          key = credential) %>%
  as_tibble()
```

```
## # A tibble: 1,181 x 5
##    state county tract  B01001_002E B01001_026E
##    <chr> <chr>  <chr>        <dbl>       <dbl>
##  1 01    073    005701        1230        1232
##  2 01    073    010704        2064        2929
##  3 01    073    012908        2782        3266
##  4 01    073    005302        1778        1850
##  5 01    073    011107        5728        5807
##  6 01    073    013300        1543        1798
##  7 01    073    014406        3015        2869
##  8 01    073    013801         974        1093
##  9 01    023    956800        1692        2347
## 10 01    023    956900        1706        1736
## # ... with 1,171 more rows
```

Resources

- Zapier Intro to APIs
- Best practices for API packages
- Building R and Stata packages for the Education Data Portal
- How we built the API for the Education Data Portal
- Why we built an API for Urban’s Education Data Portal
- Why we built the Education Data Portal
- Democratizing Big Data Processing for the Education Data Portal