Application programming interfaces

Tad Dallas

What's the point of APIs?

Application programming interfaces (or `APIs`) are used to describe the communication of one computer to a web-based source of data in a programmatic manner. The computer (client) makes a request for data in a well-documented and consistent way, and receives data from the server. This is incredibly useful for acquiring data in a programmatic way, allowing analyses to be re-run on dynamic data, creating a living analysis. This could be critical in situations such as an ongoing pandemic, where daily case count updates might change model parameters or predictions.

Many websites have APIs in order to allow users to make calls and develop apps on top of existing sites e.g., Facebook, Spotify, etc. all have APIs.

Why is this valuable? Contrast the API approach to pure web scraping. Web scraping refers to extracting data from html pages. This is different from querying an API, as APIs will have documentation about how the data are represented. For instance, what happens when a website decides to change a bit of its formatting? This would potentially break any web scraping code, but the API backend would be maintained.

In the context of biological data, many data repositories (figshare, data dryad, dataONE) have clear APIs. Further, it is becoming more common for large museum and government datasets to have developed APIs. I am a huge fan of this, as it is a departure from the longstanding "my data" mentality that has lead to a clear power dynamic (e.g., being associated with labs with lots of data gives them a clear advantage... but those data were likely gathered with a combination of taxpayer dollars and researcher sweat. The dollars trump the sweat, in my opinion.).

How do I query an API?

APIs can be queried programmtically using a number of `R` packages. Some APIs will promote their own `R` packages that have functions specific to their data structures or API. However, at the core of these packages are a small number of low-level packages for querying APIs.

The general structure of an API call is the same relatively regardless of `R` package used. The parameters of an HTTP request are typically contained in the URL. For example, to return a map of Manchester using the Google Maps Static API we would submit the following request:

```
https://maps.googleapis.com/maps/api/staticmap?center=Manchester,England&zoom=13&size=600x300&maptype=r
```

The request contains: + a URL to the API endpoint (https://maps.googleapis.com/maps/api/staticmap?)
and; + a query containing the parameters of the request (center=Manchester,England&zoom=13&size=600x300&maptype=roa
In this case, we have specified the location, zoom level, size and type of map.

Web service APIs use two key HTTP verbs to enable data requests: GET and POST. A GET request is submitted within the URL with each parameter separated by an ampersand (&). A POST request is submitted in the message body which is separate from the URL. The advantage of using POST over GET requests is that there are no character limits and the request is more secure because it is not stored in the browser's cache.

There are several types of Web service APIs (e.g. XML-RPC, JSON-RPC and SOAP) but the most popular is Representational State Transfer or REST. RESTful APIs can return output as XML, JSON, CSV and

several other data formats. Each API has documentation and specifications which determine how data can be transferred.

```
install.packages(c("httr", "jsonlite"))
```

After downloading the libraries, we will be able to use them in our `R` scripts or `RMarkdown` files.

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

A simple example of an API call

Some APIs require a token, in order to identify who is accessing the data (e.g., Spotify, Facebook, etc.). Developer access is pretty easy to get when you have an account, but it is a bit too much of a hassle for demonstration purposes. So we will use an open API from the Internet Archive's *Open Library*.

We can send a request for data by using the documented API put out by the data provider, which consists of a base url (http://openlibrary.org/search.json?) with an amended query for searching the database. We use the GET verb, which is in `R` but stems from HTTP (hypertext transfer protocol). It stores all the information in an object that is structured in JSON. Below, we search the Open Library for "Vonnegut", and receive a nested list which includes information on status codes, among other relevant information.

```
von <- httr::GET('http://openlibrary.org/search.json?q=vonnegut')
str(von)
```

```
## List of 10
##  $ url        : chr "http://openlibrary.org/search.json?q=vonnegut"
##  $ status_code: int 200
##  $ headers    :List of 10
##   ..$ server                      : chr "nginx/1.28.0"
##   ..$ date                        : chr "Thu, 15 May 2025 20:30:17 GMT"
##   ..$ content-type                : chr "application/json"
##   ..$ transfer-encoding           : chr "chunked"
##   ..$ connection                  : chr "keep-alive"
##   ..$ access-control-allow-origin: chr "*"
##   ..$ access-control-allow-method: chr "GET, OPTIONS"
##   ..$ access-control-max-age     : chr "86400"
##   ..$ x-ol-stats                  : chr "\"SR 1 0.039 TT 0 0.050\""
##   ..$ referrer-policy             : chr "no-referrer-when-downgrade"
##   ..- attr(*, "class")= chr [1:2] "insensitive" "list"
##  $ all_headers:List of 1
##   ..$ :List of 3
##   .. ..$ status : int 200
##   .. ..$ version: chr "HTTP/1.1"
##   .. ..$ headers:List of 10
##   .. .. ..$ server                      : chr "nginx/1.28.0"
##   .. .. ..$ date                        : chr "Thu, 15 May 2025 20:30:17 GMT"
##   .. .. ..$ content-type                : chr "application/json"
##   .. .. ..$ transfer-encoding           : chr "chunked"
##   .. .. ..$ connection                  : chr "keep-alive"
##   .. .. ..$ access-control-allow-origin: chr "*"
##   .. .. ..$ access-control-allow-method: chr "GET, OPTIONS"
##   .. .. ..$ access-control-max-age     : chr "86400"
##   .. .. ..$ x-ol-stats                  : chr "\"SR 1 0.039 TT 0 0.050\""
##   .. .. ..$ referrer-policy             : chr "no-referrer-when-downgrade"
##   .. .. ..- attr(*, "class")= chr [1:2] "insensitive" "list"
```

```
##  $ cookies    :'data.frame': 0 obs. of  7 variables:
##   ..$ domain    : logi(0)
##   ..$ flag      : logi(0)
##   ..$ path      : logi(0)
##   ..$ secure    : logi(0)
##   ..$ expiration: 'POSIXct' num(0)
##   ..$ name      : logi(0)
##   ..$ value     : logi(0)
##  $ content    : raw [1:80069] 7b 0a 20 20 ...
##  $ date       : POSIXct[1:1], format: "2025-05-15 20:30:17"
##  $ times      : Named num [1:6] 0 0.0203 0.0953 0.0953 9.2223 ...
##   ..- attr(*, "names")= chr [1:6] "redirect" "namelookup" "connect" "pretransfer" ...
##  $ request    :List of 7
##   ..$ method    : chr "GET"
##   ..$ url       : chr "http://openlibrary.org/search.json?q=vonnegut"
##   ..$ headers   : Named chr "application/json, text/xml, application/xml, */*"
##   .. ..- attr(*, "names")= chr "Accept"
##   ..$ fields    : NULL
##   ..$ options   :List of 2
##   .. ..$ useragent: chr "libcurl/7.81.0 r-curl/6.0.1 httr/1.4.7"
##   .. ..$ httpget  : logi TRUE
##   ..$ auth_token: NULL
##   ..$ output    : list()
##   .. ..- attr(*, "class")= chr [1:2] "write_memory" "write_function"
##   ..- attr(*, "class")= chr "request"
##  $ handle     :Class 'curl_handle' <externalptr>
##  - attr(*, "class")= chr "response"
```

This data structure is basically in JSON format, which is a different file format (e.g., xml, html, etc.). To make it something that `R` can work with, we use the `jsonlite` package, and function `fromJSON`.

```
vonInfo <- jsonlite::fromJSON(content(von, "text"), simplifyVector = FALSE)
```

```
## No encoding supplied: defaulting to UTF-8.
```

This output is a list of length 4, each containing a nested list. It is not the cleanest output, but it rarely is. But all the information you could want is buried somewhere here.

This apply statement gets information on the title of each of the works returned from our search for "Vonnegut".

```
vonTitle0 <- sapply(vonInfo[[8]], function(x){
  x$title
})
```

But Vonnegut as a search term is misleading, because many people have written books on Vonnegut, while we may want books *by* Vonnegut.

```
von2 <- httr::GET('http://openlibrary.org/search.json?author=vonnegut')
str(von2)
```

```
## List of 10
##  $ url        : chr "http://openlibrary.org/search.json?author=vonnegut"
##  $ status_code: int 200
##  $ headers    :List of 10
##   ..$ server                    : chr "nginx/1.28.0"
##   ..$ date                      : chr "Thu, 15 May 2025 20:30:27 GMT"
##   ..$ content-type              : chr "application/json"
##   ..$ transfer-encoding         : chr "chunked"
```

```
##   ..$ connection                : chr "keep-alive"
##   ..$ access-control-allow-origin: chr "*"
##   ..$ access-control-allow-method: chr "GET, OPTIONS"
##   ..$ access-control-max-age    : chr "86400"
##   ..$ x-ol-stats                : chr "\"SR 1 0.108 TT 0 0.171\""
##   ..$ referrer-policy           : chr "no-referrer-when-downgrade"
##   ..- attr(*, "class")= chr [1:2] "insensitive" "list"
##  $ all_headers:List of 1
##   ..$ :List of 3
##   .. ..$ status : int 200
##   .. ..$ version: chr "HTTP/1.1"
##   .. ..$ headers:List of 10
##   .. .. ..$ server                     : chr "nginx/1.28.0"
##   .. .. ..$ date                       : chr "Thu, 15 May 2025 20:30:27 GMT"
##   .. .. ..$ content-type               : chr "application/json"
##   .. .. ..$ transfer-encoding          : chr "chunked"
##   .. .. ..$ connection                 : chr "keep-alive"
##   .. .. ..$ access-control-allow-origin: chr "*"
##   .. .. ..$ access-control-allow-method: chr "GET, OPTIONS"
##   .. .. ..$ access-control-max-age     : chr "86400"
##   .. .. ..$ x-ol-stats                 : chr "\"SR 1 0.108 TT 0 0.171\""
##   .. .. ..$ referrer-policy            : chr "no-referrer-when-downgrade"
##   .. .. ..- attr(*, "class")= chr [1:2] "insensitive" "list"
##  $ cookies    :'data.frame': 0 obs. of  7 variables:
##   ..$ domain    : logi(0)
##   ..$ flag      : logi(0)
##   ..$ path      : logi(0)
##   ..$ secure    : logi(0)
##   ..$ expiration: 'POSIXct' num(0)
##   ..$ name      : logi(0)
##   ..$ value     : logi(0)
##  $ content    : raw [1:95264] 7b 0a 20 20 ...
##  $ date       : POSIXct[1:1], format: "2025-05-15 20:30:27"
##  $ times      : Named num [1:6] 0 0.000029 0.000029 0.000061 9.284609 ...
##   ..- attr(*, "names")= chr [1:6] "redirect" "namelookup" "connect" "pretransfer" ...
##  $ request    :List of 7
##   ..$ method    : chr "GET"
##   ..$ url       : chr "http://openlibrary.org/search.json?author=vonnegut"
##   ..$ headers   : Named chr "application/json, text/xml, application/xml, */*"
##   .. ..- attr(*, "names")= chr "Accept"
##   ..$ fields    : NULL
##   ..$ options   :List of 2
##   .. ..$ useragent: chr "libcurl/7.81.0 r-curl/6.0.1 httr/1.4.7"
##   .. ..$ httpget  : logi TRUE
##   ..$ auth_token: NULL
##   ..$ output    : list()
##   .. ..- attr(*, "class")= chr [1:2] "write_memory" "write_function"
##   ..- attr(*, "class")= chr "request"
##  $ handle     :Class 'curl_handle' <externalptr>
##  - attr(*, "class")= chr "response"
```

This data structure is basically in JSON format, which is a different file format (e.g., xml, htm, etc.). To make it something that `R` can work with, we use the `jsonlite` package, and function `fromJSON`.

```
vonInfo2 <- jsonlite::fromJSON(content(von2, "text"), simplifyVector = FALSE)
```

```
## No encoding supplied: defaulting to UTF-8.
```

This is better, as it now provides information on books where Vonnegut was listed as an author.

```
vonTitle <- sapply(vonInfo2[[8]], function(x){
  x$title
})
```

Side fun note: You can access GitHub info through API calls as well

```
tad <- httr::GET('https://api.github.com/users/taddallas')
tadInfo <- jsonlite::fromJSON(content(tad, "text"), simplifyVector = FALSE)
```

Class problem:

How long as Tad been on GitHub?

A more biological example

So this gives a good background about APIs, but it does not give a good biological example of when they are useful. The books that Vonnegut wrote are unlikely to change, so the search queries to this API are fairly static, and a bit removed from biology.

One good biology API that is open is the Global Biodiversity Information Facility (GBIF). We can use a similar structure to our Open Library query to obtain information on species names and occurrences.

```
giraffe <- httr::GET('https://www.gbif.org/developer/species/Giraffa+camelopardalis')
str(giraffe)
```

```
## List of 10
##  $ url        : chr "https://www.gbif.org/developer/species/Giraffa+camelopardalis"
##  $ status_code: int 404
##  $ headers    :List of 19
##   ..$ content-security-policy          : chr "default-src maxcdn.bootstrapcdn.com cdn.jsdelivr.net/c
##   ..$ x-dns-prefetch-control           : chr "off"
##   ..$ expect-ct                        : chr "max-age=0"
##   ..$ x-frame-options                  : chr "SAMEORIGIN"
##   ..$ strict-transport-security        : chr "max-age=600; includeSubDomains"
##   ..$ x-download-options               : chr "noopen"
##   ..$ x-content-type-options           : chr "nosniff"
##   ..$ x-permitted-cross-domain-policies: chr "none"
##   ..$ x-xss-protection                 : chr "0"
##   ..$ x-request-id                     : chr "70321d60-31cb-11f0-ad43-538b2798c058"
##   ..$ cache-control                    : chr "public, max-age=60, must-revalidate"
##   ..$ content-type                     : chr "text/html; charset=utf-8"
##   ..$ etag                             : chr "W/\"efbb-ChB23/2b3mxJqR54rJwhKMt1ls0\""
##   ..$ vary                             : chr "Accept-Encoding"
##   ..$ content-encoding                 : chr "gzip"
##   ..$ date                             : chr "Thu, 15 May 2025 20:30:27 GMT"
##   ..$ x-varnish                        : chr "264997191"
##   ..$ age                              : chr "0"
##   ..$ via                              : chr "1.1 varnish (Varnish/6.6)"
##   ..- attr(*, "class")= chr [1:2] "insensitive" "list"
##  $ all_headers:List of 1
##   ..$ :List of 3
```

```
##   .. ..$ status : int 404
##   .. ..$ version: chr "HTTP/2"
##   .. ..$ headers:List of 19
##   .. .. ..$ content-security-policy         : chr "default-src maxcdn.bootstrapcdn.com cdn.jsdelivr
##   .. .. ..$ x-dns-prefetch-control          : chr "off"
##   .. .. ..$ expect-ct                       : chr "max-age=0"
##   .. .. ..$ x-frame-options                 : chr "SAMEORIGIN"
##   .. .. ..$ strict-transport-security       : chr "max-age=600; includeSubDomains"
##   .. .. ..$ x-download-options              : chr "noopen"
##   .. .. ..$ x-content-type-options          : chr "nosniff"
##   .. .. ..$ x-permitted-cross-domain-policies: chr "none"
##   .. .. ..$ x-xss-protection                : chr "0"
##   .. .. ..$ x-request-id                    : chr "70321d60-31cb-11f0-ad43-538b2798c058"
##   .. .. ..$ cache-control                   : chr "public, max-age=60, must-revalidate"
##   .. .. ..$ content-type                    : chr "text/html; charset=utf-8"
##   .. .. ..$ etag                            : chr "W/\"efbb-ChB23/2b3mxJqR54rJwhKMt1ls0\""
##   .. .. ..$ vary                            : chr "Accept-Encoding"
##   .. .. ..$ content-encoding                : chr "gzip"
##   .. .. ..$ date                            : chr "Thu, 15 May 2025 20:30:27 GMT"
##   .. .. ..$ x-varnish                       : chr "264997191"
##   .. .. ..$ age                             : chr "0"
##   .. .. ..$ via                             : chr "1.1 varnish (Varnish/6.6)"
##   .. .. ..- attr(*, "class")= chr [1:2] "insensitive" "list"
##  $ cookies    :'data.frame': 0 obs. of  7 variables:
##   ..$ domain    : logi(0)
##   ..$ flag      : logi(0)
##   ..$ path      : logi(0)
##   ..$ secure    : logi(0)
##   ..$ expiration: 'POSIXct' num(0)
##   ..$ name      : logi(0)
##   ..$ value     : logi(0)
##  $ content    : raw [1:61371] 0a 3c 21 44 ...
##  $ date       : POSIXct[1:1], format: "2025-05-15 20:30:27"
##  $ times      : Named num [1:6] 0 0.0416 0.1613 0.3154 0.4503 ...
##   ..- attr(*, "names")= chr [1:6] "redirect" "namelookup" "connect" "pretransfer" ...
##  $ request    :List of 7
##   ..$ method    : chr "GET"
##   ..$ url       : chr "https://www.gbif.org/developer/species/Giraffa+camelopardalis"
##   ..$ headers   : Named chr "application/json, text/xml, application/xml, */*"
##   .. ..- attr(*, "names")= chr "Accept"
##   ..$ fields    : NULL
##   ..$ options   :List of 2
##   .. ..$ useragent: chr "libcurl/7.81.0 r-curl/6.0.1 httr/1.4.7"
##   .. ..$ httpget  : logi TRUE
##   ..$ auth_token: NULL
##   ..$ output    : list()
##   .. ..- attr(*, "class")= chr [1:2] "write_memory" "write_function"
##   ..- attr(*, "class")= chr "request"
##  $ handle     :Class 'curl_handle' <externalptr>
##  - attr(*, "class")= chr "response"
```

Before we dive too far down this hole, it is important to note that this wheel has already been invented and is being maintained by the ROpenSci collective, a group of `R` programmers who develop tools for the access and analysis of data resources. We will use this package, not because it enhances reproducibility (as it is

yet another dependency), but because it is well-written, well-documented, and from a group of scientists committed to helping make open data available and analyses reproducible.

```
# install.packages('rgbif')
library(rgbif)
```

```
giraffe <- rgbif::occ_search(scientificName = "Giraffa camelopardalis",
    limit = 5000)
```

> Depending on your operating system (OS) and previously installed packages, this is where we start to get into issues of dependencies. The `rgbif` package requires the installation of `rgeos`, which requires the `geos` library to be installed outside of `R`. Hopefully you will be able to successfully install the package, but it is an important note that handling these dependency and OS-specific issues is pretty central to making sure an analytical pipeline is reproducible.

So presumably we have successfully queried the GBIF API using the `rgbif` package, and now have a data.frame of giraffe occurrences (limited to 5000 occurrence points). The output data are structured as a list of lists, which is a bit confusing if we look at the `str()` of `giraffe`, as the authors of the package have created a `class` called `gbif` to hold all the data. This is common in `R` packages, and the authors have written in some great functionality in printing and working with these data, formatting the data as a `tibble` object. We will not go too much into `tibble`s, but they are pretty useful for keeping data in a "tidy" format, as columns in a tibble can be lists (e.g., what if we wanted to store spatial polygon information, vectors, etc. alongside each row element? `tibble` handles this need effectively).

```
typeof(giraffe)
```

```
## [1] "list"
```

```
class(giraffe)
```

```
## [1] "gbif"
```

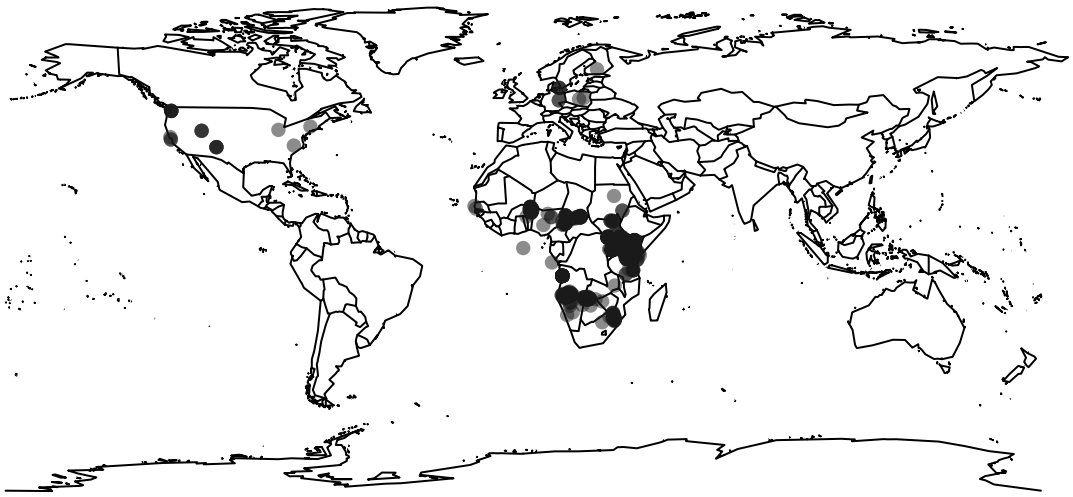
For simplicity sake, we can use some of the functionality within `rgbif` to just return the data we are interested in, specifying that we only want the raw data, not all the potential data (which includes images, etc.)

```
giraffe2 <- rgbif::occ_search(scientificName = "Giraffa camelopardalis",
    limit = 5000, return='data')[[3]]
```

```
## Warning in pchk(return, "occ_search"): `return` param in `occ_search` function is defunct as of rgbi
## See `?rgbif` for more information.
```

We will quickly plot these data out just to show them in geographic space. We first create a base map using the `maps` package, which is a really easy way to plot geopolitical boundaries. Then we layer on the occurrence data using the `points` function.

```
# install.packages('maps')

maps::map()
points(giraffe2$decimalLongitude, giraffe2$decimalLatitude, pch=16,
    col=grey(0.1,0.5))
```



Giraffes are in the US!!! It is important to clean your data, as these are 100% zoo records.

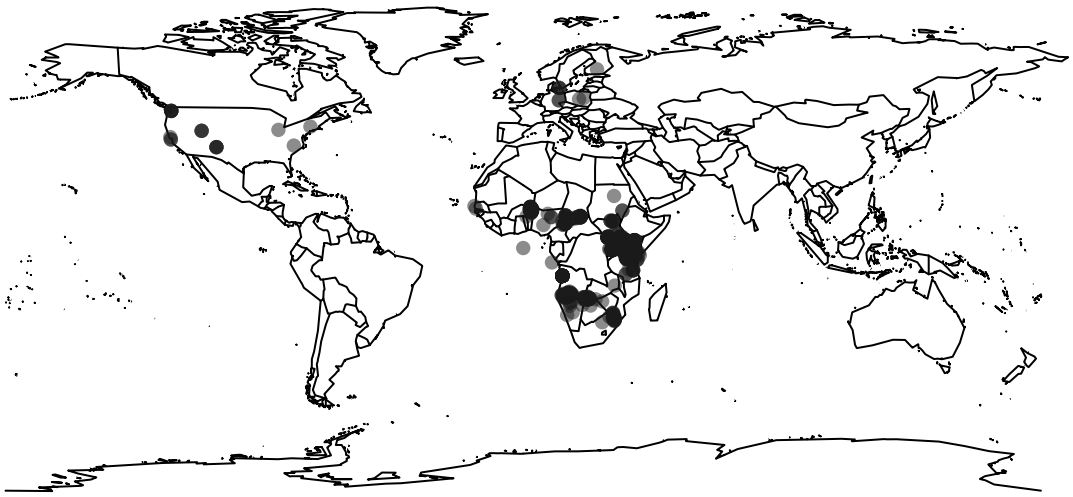
A fun aside on spatial data in R

A fairly consistent ask of researchers who use `R` is how to make pretty maps. There are a number of different `R` packages which can help you work with spatial data, and a full exploration of these packages is outside the scope of this course. However, I will point you to a few resources that I have found useful.

- R-spatial
- R spatial task view
- R-spatial book

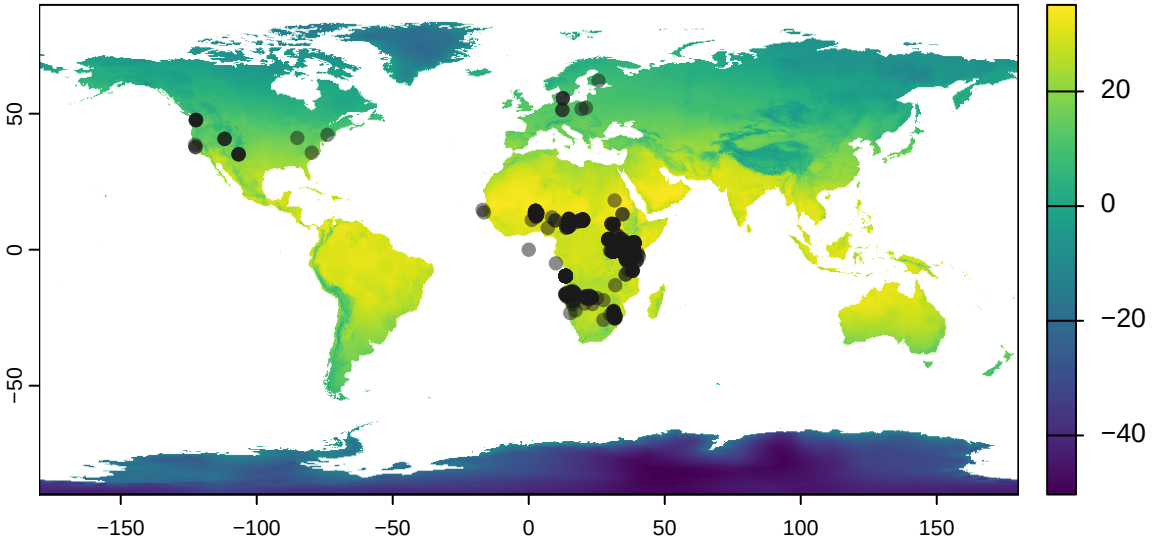
For visualizing spatial boundaries, I have found the `maps` package to be the most useful, though the `geodata` package also has world boundaries at different 'resolutions' or levels of detail. A short time ago, the main `R` packages for spatial data were `raster` and `sp`, both have largely been refined and replaced by `terra` and `sf`, respectively (same maintainers for the most part as well).

```
library(sf)
giraffe2 <- giraffe2 %>%
  dplyr::select(decimalLongitude, decimalLatitude) %>%
  dplyr::filter(!is.na(decimalLongitude), !is.na(decimalLatitude))

pts <- sf::st_as_sf(giraffe2, coords = c("decimalLongitude", "decimalLatitude"),
  agr = "constant")

maps::map()
plot(pts, add = TRUE, pch = 16, col = grey(0.1,0.5))
```



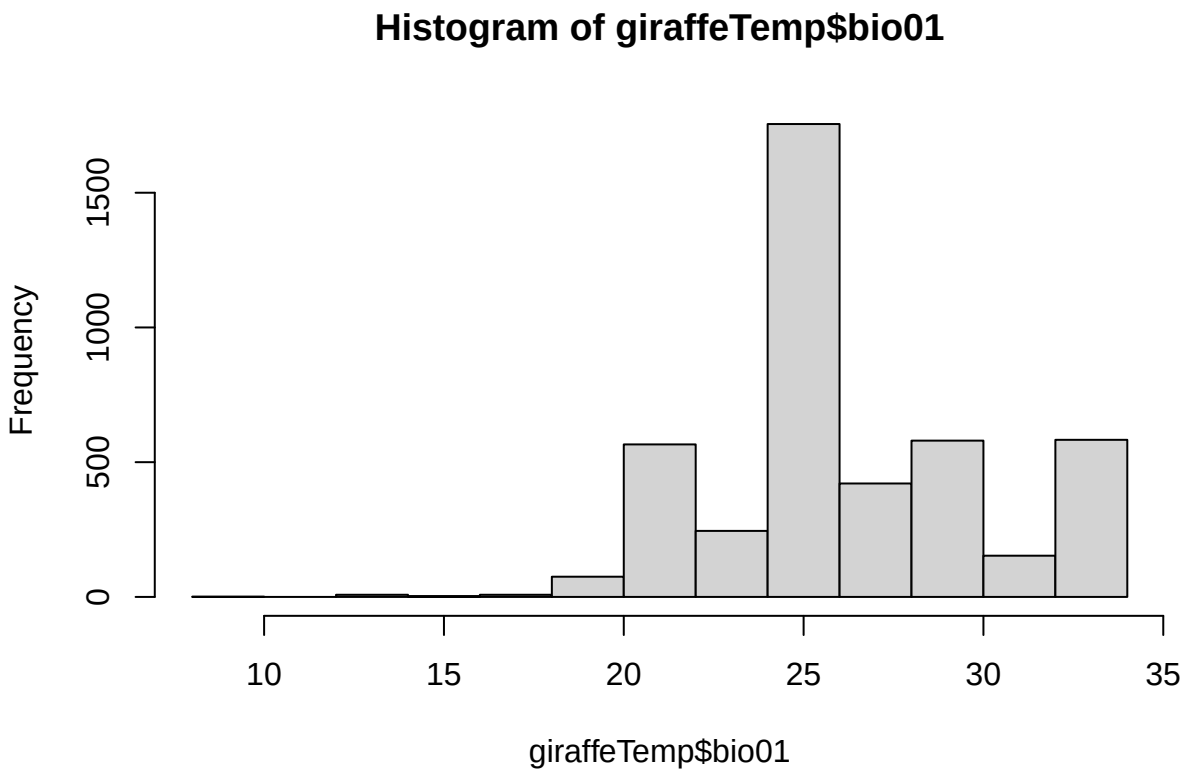
Above is a good example of how to convert points from a data.frame into a spatial object that `sf` can work with (it needs information on things like projection and other metadata). Now let's go into another spatial data type, the raster, and how we can use point data alongside raster data.

```
library(terra)
library(geodata)

bc <- geodata::cmip6_world("CNRM-CM6-1", "585", "2061-2080", var="bioc", res=10, path=tempdir())

plot(bc[[1]])
plot(pts, add = TRUE, pch = 16, col = grey(0.1,0.5))
```



And to get the values of the raster at the points we have, we can use the `extract` function from `terra`. The `extract` function will take the values from the raster at the points you provide, and return a data.frame with the values.

```
giraffeTemp <- terra::extract(bc[[1]], pts)
hist(giraffeTemp$bio01)
```



Interfacing with local databases

Accessing data through APIs is fantastic, but there are plenty of times where you may need to access large data files locally. Note that this only is necessary if the data you are working with cannot fit into local memory. Otherwise, there is no real advantage to using a database framework, as it will likely be more expensive in terms of time and potential frustration.

For this, we will have to include yet another dependency (5+ in this lecture alone), which of course has a bunch of other dependencies. This is not really ideal given the focus on reproducible research in the course, but it is sadly pretty much necessary.

```
install.packages('dbplyr')
```

When referring to "databases", we may think that there is a single type or schema, but this is definitely not the case. `R` has a way to accommodate for different database types, and that is to include a dependency for each database type (maybe not ideal). Below is a list describing the different ways `R` interfaces with different database types (taken from https://www.kaggle.com/anello/working-with-databases-in-r).

- RMySQL connects to MySQL and MariaDB
- RPostgreSQL connects to Postgres and Redshift.
- RSQLite embeds a SQLite database.
- odbc connects to many commercial databases via the open database connectivity protocol.
- bigrquery connects to Google's BigQuery.

We will focus on the use of RSQLite as a backend to connect to SQLite databases. This will sadly require another dependency.

```
install.packages('RSQLite')
```

```
dir.create("data_raw", showWarnings = FALSE)
download.file(url = "https://ndownloader.figshare.com/files/2292171",
              destfile = "data_raw/portal_mammals.sqlite", mode = "wb")
```

```
library(DBI)

mammals <- DBI::dbConnect(RSQLite::SQLite(), "data_raw/portal_mammals.sqlite")
dbplyr::src_dbi(mammals)
```

```
## src:  sqlite 3.46.0 [/media/tad/sanDisk1TB/Teaching/biologicalDataScience/website/content/code/data_
## tbls: plots, species, surveys
```

Just like a spreadsheet with multiple worksheets, a SQLite database can contain multiple tables. In this case three of them are listed in the tbls row in the output above:

- plots
- species
- surveys

Now that we know we can connect to the database, let's explore how to get the data from its tables into R.

Querying the database with the SQL syntax

To connect to tables within a database, you can use the `tbl()` function from `dplyr`. This function can be used to send SQL queries to the database. To demonstrate this functionality, let's select the columns "year", "species_id", and "plot_id" from the surveys table:

```
dplyr::tbl(mammals, dplyr::sql("SELECT year, species_id, plot_id FROM surveys"))
```

```
## # Source:   SQL [?? x 3]
## # Database: sqlite 3.46.0 [/media/tad/sanDisk1TB/Teaching/biologicalDataScience/website/content/code/
##     year species_id plot_id
##    <int> <chr>        <int>
##  1  1977 NL               2
##  2  1977 NL               3
##  3  1977 DM               2
##  4  1977 DM               7
##  5  1977 DM               3
##  6  1977 PF               1
##  7  1977 PE               2
##  8  1977 DM               1
##  9  1977 DM               1
## 10  1977 PF               6
## # i more rows
```

With this approach you can use any of the SQL queries we have seen in the database lesson.

Querying the database with the dplyr syntax

One of the strengths of dplyr is that the same operation can be done using dplyr verbs instead of writing SQL. First, we select the table on which to do the operations by creating the surveys object, and then we use the standard dplyr syntax as if it were a data frame:

```
surveys <- dplyr::tbl(mammals, "surveys")
surveys %>%
    dplyr::select(year, species_id, plot_id)
```

```
## # Source:   SQL [?? x 3]
## # Database: sqlite 3.46.0 [/media/tad/sanDisk1TB/Teaching/biologicalDataScience/website/content/code/
##     year species_id plot_id
##    <int> <chr>        <int>
##  1  1977 NL               2
```

```
##  2  1977 NL              3
##  3  1977 DM              2
##  4  1977 DM              7
##  5  1977 DM              3
##  6  1977 PF              1
##  7  1977 PE              2
##  8  1977 DM              1
##  9  1977 DM              1
## 10  1977 PF              6
## # i more rows
```

In this case, the surveys object behaves like a data frame. Several functions that can be used with data frames can also be used on tables from a database. For instance, the `head()` function can be used to check the first 10 rows of the table:

```
head(surveys, n = 10)
```

```
## # Source:   SQL [?? x 9]
## # Database: sqlite 3.46.0 [/media/tad/sanDisk1TB/Teaching/biologicalDataScience/website/content/code/
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <int> <int> <int> <int>   <int> <chr>      <chr>           <int>  <int>
##  1         1     7    16  1977       2 NL         M                  32     NA
##  2         2     7    16  1977       3 NL         M                  33     NA
##  3         3     7    16  1977       2 DM         F                  37     NA
##  4         4     7    16  1977       7 DM         M                  36     NA
##  5         5     7    16  1977       3 DM         M                  35     NA
##  6         6     7    16  1977       1 PF         M                  14     NA
##  7         7     7    16  1977       2 PE         F                  NA     NA
##  8         8     7    16  1977       1 DM         M                  37     NA
##  9         9     7    16  1977       1 DM         F                  34     NA
## 10        10     7    16  1977       6 PF         F                  20     NA
```

```
surveys2 <- surveys %>%
  dplyr::filter(weight < 5) %>%
  dplyr::select(species_id, sex, weight)
```

`R` makes lazy calls to databases, until you make it not be lazy. That is, everything is held outside of memory until you tell `R` that some portion of the data should not be.

```
head(surveys2)
```

```
## # Source:   SQL [?? x 3]
## # Database: sqlite 3.46.0 [/media/tad/sanDisk1TB/Teaching/biologicalDataScience/website/content/code/
##   species_id sex   weight
##   <chr>      <chr>  <int>
## 1 PF         M          4
## 2 PF         F          4
## 3 PF         <NA>       4
## 4 PF         F          4
## 5 PF         F          4
## 6 RM         M          4
```

To pull the data into memory and allow us to use the data, we must use the `collect` function.

```
surveys3 <- dplyr::collect(surveys2)
head(surveys3)
```

```
## # A tibble: 6 x 3
```

```
##   species_id sex   weight
##   <chr>      <chr>  <int>
## 1 PF         M          4
## 2 PF         F          4
## 3 PF         <NA>       4
## 4 PF         F          4
## 5 PF         F          4
## 6 RM         M          4
```

This difference becomes clear when we try to index a specific column of data and work with it.

```
surveys2$sex
surveys3$sex
```

Relating this back to the `dplyr` syntax of joins

We discussed joins in the data manipulation section. Joins are pretty key to working with databases, as much of the benefit of database structure is from nested data and the utility of key values. For instance, in our mammal sampling data, we have plot level data...

```
plots <- dplyr::tbl(mammals, "plots")
plots
```

```
## # Source:   table<`plots`> [?? x 2]
## # Database: sqlite 3.46.0 [/media/tad/sanDisk1TB/Teaching/biologicalDataScience/website/content/code/
##    plot_id plot_type
##      <int> <chr>
##  1       1 Spectab exclosure
##  2       2 Control
##  3       3 Long-term Krat Exclosure
##  4       4 Control
##  5       5 Rodent Exclosure
##  6       6 Short-term Krat Exclosure
##  7       7 Rodent Exclosure
##  8       8 Control
##  9       9 Spectab exclosure
## 10      10 Rodent Exclosure
## # i more rows
```

and survey-level data, where plot-level data provides what is essentially metadata for each survey.

We can use joins to combine and manipulate these data, as we did before, but now in the context of the database structure.

```
survPlot <- dplyr::left_join(surveys, plots, by='plot_id')
```

sessionInfo

```
sessionInfo()
```

```
## R version 4.5.0 (2025-04-11)
## Platform: x86_64-pc-linux-gnu
## Running under: Ubuntu 22.04.5 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/atlas/libblas.so.3.10.3
## LAPACK: /usr/lib/x86_64-linux-gnu/atlas/liblapack.so.3.10.3;  LAPACK version 3.10.0
```

```
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## time zone: America/New_York
## tzcode source: system (glibc)
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] DBI_1.2.3      geodata_0.5-8  terra_1.8-50   sf_1.0-21      rgbif_3.7.7
## [6] jsonlite_1.8.9 httr_1.4.7
##
## loaded via a namespace (and not attached):
##  [1] gtable_0.3.6       xfun_0.49          servr_0.27         ggplot2_3.5.1
##  [5] processx_3.8.4     vctrs_0.6.5        tools_4.5.0        ps_1.8.1
##  [9] generics_0.1.4     curl_6.0.1         RSQLite_2.3.7      tibble_3.2.1
## [13] proxy_0.4-27       blob_1.2.4         pkgconfig_2.0.3    KernSmooth_2.23-26
## [17] data.table_1.16.2  dbplyr_2.5.0       lifecycle_1.0.4    compiler_4.5.0
## [21] stringr_1.5.1      munsell_0.5.1      tinytex_0.54       codetools_0.2-19
## [25] httpuv_1.6.15      htmltools_0.5.8.1  maps_3.4.1         class_7.3-23
## [29] yaml_2.3.10        lazyeval_0.2.2     later_1.4.1        pillar_1.10.2
## [33] whisker_0.4.1      classInt_0.4-11    cachem_1.1.0       tidyselect_1.2.1
## [37] digest_0.6.37      stringi_1.8.4      purrr_1.0.2        dplyr_1.1.4
## [41] fastmap_1.2.0      grid_4.5.0         colorspace_2.1-1   cli_3.6.5
## [45] magrittr_2.0.3     utf8_1.2.5         triebeard_0.4.1    crul_1.5.0
## [49] e1071_1.7-16       withr_3.0.2        scales_1.3.0       promises_1.3.2
## [53] bit64_4.5.2        oai_0.4.0          rmarkdown_2.29     bit_4.5.0
## [57] blogdown_1.18      memoise_2.0.1      evaluate_1.0.1     knitr_1.49
## [61] rlang_1.1.6        urltools_1.7.3     Rcpp_1.0.14        glue_1.8.0
## [65] httpcode_0.3.0     xml2_1.3.6         rstudioapi_0.17.1  R6_2.6.1
## [69] plyr_1.8.9         units_0.8-7
```