Web Mapping and Analysis

Elisabetta Pietrostefani and Dani Arribas-Bel

3/13/23

Table of contents

Welcome

This is the website for "Web Mapping and Analysis" for the module **ENVS456** at the University of Liverpool. This is course designed and delivered by Dr. Elisabetta Pietrostefani and Professor Dani Arribas-Bel from the Geographic Data Science Lab at the University of Liverpool, United Kingdom. The module has two main aims. First, it seeks to to provide hands-on experience and training in the design and generation of web-based mapping and geographical information tools. Second, it seeks to provide hands-on experience and training in the use of software to access, analyse and visualize web-based geographical information.

 A compilation of this web course is hosted as a GitHub repository that you can access:

- As an html website.
- As a pdf document
- As a GitHub repository.

Contact

Dani Arribas-Bel - darribas [at] liverpool.ac.uk Professor in Geographic Data Science Office 6xx, Roxby Building, University of Liverpool - 74 Bedford St S, Liverpool, L69 7ZT, United Kingdom.

Elisabetta Pietrostefani - e.pietrostefani [at] liverpool.ac.uk Lecturer in Geographic Data Science Office 6xx, Roxby Building, University of Liverpool - 74 Bedford St S, Liverpool, L69 7ZT, United Kingdom.

Syllabus

Introduction

Block 1

- Lecture: Introduction to the module
- Lab: Powerful examples

Data Backends

Block 2

- Lecture: The Web's architecture and Economy
- Lab: What do APIs actually do?

Block 3

- Lecture: Data architechtures & formats
- Lab: Creating, manipulating, and integrating web geospatial data

Block 4

- Lecture: APIs
- Lab: Acquiring data from the web

Assignment I

Block 5

- Lecture: Q&A
- Lab: Clinic

`Assignment I`: Combining (geo-)data in an interactive map

Frontend Topics

Block 6

- Lecture: Map design
- Lab: Designing maps with Kepler

Block 7

- Lecture: Interactivity
- Lab: Designing for interactivity

Block 8

- Lecture: Statistical visualisation
- Lab: Choropleths in Kepler

Dashboards

Block 9

- Lecture: Dashboards: bringing analysis to the web
- Lab: Building Dashboards (Shiny)

Block 10

- Lecture: Technology gallery
- Lab: Assignment II clinic

`Assignment II`: A dashboard of IMD

Overview

Aims

This module aims to:

- Provide hands-on experience and training in the design and generation of web-based mapping and geographical information tools.
- Provide hands-on experience and training in the use of software to access, analyse and visualize web-based geographical information.

Learning Outcomes

By the end of the module, students should be able to:

(1) Experience using tile rendering tools to generate content for map-based web sites.

(2) Knowledge of web based mapping infrastructure

(3) Web-based data collection techniques (accessing Twitter, Facebook, Google and Open-Streetmap information)

(4) Network analysis

(5) Programming skills to enable basic online data manipulation and web mapping

Feedback

Formal assessment. Two pieces of coursework (50%/50%). Equivalent to 2,500 words each

Verbal face-to-face feedback. Immediate face-to-face feedback will be provided during computer, discussion and clinic sessions in interaction with staff. This will take place in all live sessions during the semester.

Teams Forum. Asynchronous written feedback will be provided via Teams. Students are encouraged to contribute by asking and answering questions relating to the module content. Staff will monitor the forum Monday to Friday 9am-5pm, but it will be open to students to

make contributions at all times. Response time will vary depending on the complexity of the question and staff availability.

Computational Environment

This course can be followed by anyone with access to a bit of technical infrastructure. This section details the set of local and online requirements you will need to be able to follow along, as well as instructions or pointers to get set up on your own. This is a centralized section that lists *everything* you will require, but keep in mind that different blocks do not always require everything all the time.

To reproduce the code in the book, you need the most recent version of R and packages. These can be installed following the instructions provided in our R installation guide.

Software

To run the analysis and reproduce the code, you need the following software:

- QGIS- the stable version (3.22 LTR at the time of writing) is OK, any more recent version will also work.
- R-4.2.2
- RStudio 2022.12.0-353
- Quarto 1.2.280
- the list of libraries in the next section

To install and update:

- QGIS, download the appropriate version from QGIS.org
- R, download the appropriate version from The Comprehensive R Archive Network (CRAN)
- RStudio, download the appropriate version from Posit
- Quarto, download the appropriate version from the Quarto website

To check your version of:

- R and libraries run `sessionInfo()`
- RStudio click `help` on the menu bar and then `About`
- Quarto check the `version` file in the quarto folder on your computer.

R List of libraries

The list of libraries used in this book is provided below:

- `sf`
- `geojsonsf`
- `mapview`
- `tidyverse`
- `tidycensus`
- `viridis`
- `viridisLite`
- `httr`
- `jsonlite`
- `plyr`
- `wellknown`
- `leaflet`
- `mapboxapi`
- `mapdeck`
- `stplanr`
- `classInt`
- `tmap`
- `tidygeocoder`

Online accounts

- CDRC Data: we will use some of the data provided by the CDRC, so a (free) account with them will be necessary.
- Mapbox: Mapbox is one of the industry leaders in web mapping. Their free tier is rather generous so will more than suffice for what we will do within the course. You can sign up for a new (free) account here.
- Kepler: is a data agnostic, WebGL empowered, high-performance web application for geospatial analytic visualizations.
- Carto: CARTO provides an online platform for web mapping and location intelligence. Please sign up for CARTO for Education. Please follow the steps here. Do not sign up for the free account. There are 4 steps
  - Step 1: Sign up for Github
  - Step 2: Apply for the Github Education Pack

- Step 3: Wait for verification and confirm. **This could take from 1 hour to several days. Please be patient and wait for your official verification, it is important for the process.**
- Step 4: Claim your CARTO student account

Assessments

Assignment I

- **Title:** Exploring APIs in R
- **Type:** Coursework
- Due date: **Thursday March 2nd, Week 5**
- 50% of the final mark
- Chance to be reassessed
- Electronic submission only

In this assessment, you will have the opportunity to explore different sources and combine them in a single tileset that can be explored interactively through a web browser. **The assignment aims to evaluate your knowledge and aptitude in the following areas**:

- Understanding of core "backend" concepts in web mapping such as tilesets, client-server architecture, or APIs.
- Ability to use the web as a resource for original data.
- Design skills to present effectively a diverse set of geospatial data in a web map.

Design, data and assemblage.

This assignment requires you to source data from the web in different formats, assemble them into a tileset, and document the process. To be successful, you will need to demonstrate your understanding not only of the technical aspects involved in the process, but also of the conceptual notions that underpin them. Below are described the required components for your submission.

First, the design. Start by designing a map for an area you are interested in. There are no clear restrictions but, to ensure you are on the right path, check on your ideas with the module leader, who will be able to assess whether potential problems may arise from your choices. This stage should draw some inspiration from the first weeks of the course, where we looked for examples of web maps and spent time discussing what made them good and why.

Second, the data. Draft a list of potential data that would be ideal to use for your map, and try to find out whether they exist and are available. This will be a good guide for which data

you will actually end up using. Do not worry about spending a significant amount of time on this aspect; identifying good data takes time and is at the core of this task. Make sure you include both data you can access from direct downloads (e.g. CDRC) and those you download through an API. Once you know which datasets you need, go ahead and do the work required to download them for the map you want to build.

Third, the assemblage. With all data you have at your disposal from the previous stage, create a tileset that allows to embed the map in an HTML file and explore it through the browser. Pay attention to the design aspects involved in this step too. For example, what is the extent of your map (not necessarily the extent of each of your data)? What are the zoom levels your map will allow? Do you have the same "map" for every zoom level? These are questions you will have to ask (and answer!) yourself to complete this stage successfully.

Presentation of your work

Once you have created your map, you will need to present it. An important aspect of this stage is that it is not really the map you need to present, but the *process* of creation you have followed and the design choices you have made that should go into the text. Additionally, you will need to provide evidence that you understand the concepts behind some of the technologies you have used. Write up to 1,000 words and include the following:

- Map brief
  - About 250 words introducing the map. This should cover what it tries to represent (what is it about?) and the choices you have made along the way to take that idea into fruition.
  - About 250 words discussing and motivating the sources of data you have used. Here you should engage not only with what data you are using but why and what they bring to the map. Everything should be in the map for a reason, make sure to spell it out clearly.
- Conceptual background
  - About 250 words with your description of what an API is, how it works and how it has made your map possible.
  - About 250 words with your description of how tile-based maps work.

Submit

Once completed, you will need to submit the following:

1. A static PDF version of an R-markdown. You can either generate this as a pdf directly in Rstudio or generate the html, open it in a browser like Google Chrome and save as a pdf. The pdf should include two parts:
    1. All your narrative about the map brief and conceptual background.
    2. Any code you may have used to complete the assignment, **documented** in detail. **NOTE**: this section will not contribute towards the word count.
2. A an .mbtile file containing you tileset.

The assignment will be evaluated based on four main pillars, on which you will have to be successful to achieve a good mark:

1. **Map design abilities.** This includes ideas that were discussed in the course in Blocks 1 and 2.
2. **Technical skills**. This includes your ability to master technologies that allow you to create a compelling map, but also to access interesting and sophiticated data sources.
3. **Overall narrative**. This assesses your aptitude to introduce, motivate and justify your map, as well as you ability to bring each component of the assignment into a coherent whole that "fits together".
4. **Conceptual understanding of key technologies** presented in the course, in particular of APIs and tile-based mapping.

How is this assignment useful?

This assessment includes several elements that will help you improve critical aspects of your web mapping skills:

Design: this is not about making maps, this is about making good maps. And behind every good map there is a set of conscious choices that you will have to think throug to be successful (what map? what data? how to present the data? etc.). Technology: at the end of the day, building good web maps requires solid understanding of current technology that goes beyond what the average person can be expected to know. In this assignment, you will need to demonstrate you are proficient in a series of tasks manipulating geospatial data in a web environment. Presentation: in many real-world contexts, your work is as good as it can come across to the audience it is intended to. This means that it is vital to be able to communicate not only what you are doing but why and on what building blocks it is based on.

Assignment II

- **Title:** *A dashboard of IMD*
- **Type:** Coursework
- Due date**: Thursday April 27th, Week 10**
- 50% of the final mark
- Chance to be reassessed
- Electronic submission only

This assignment requires you to build a dashboard for the Index of Multiple Deprivation. To be successful, you will need to demonstrate your understanding not only of technical elements, but of the design process required to create a product that can communicate complex ideas effectively. There are three core building blocks you will have to assemble to build your dashboard: basemap, main map(s), and widgets. Let us explore each of them more in detail.

First, the basemap. Design your own basemap using Mapbox. Think about the data in the background, which colors, the zoom levels that will be allowed, and how it all comes together to create a backdrop for your main message that is conducent to the experience you want to create. The basemap is like a good side dish: it's there to make you like the main course even more.

Second, the main map(s). One you have your own basemap from Mapbox… move to R. This is where the core of your dashboard should come to shine. What you want to show, how, which interactive elements you will allow the user to access and how they will let them modify the experience of your dashboard. The main course of the meal, make it count!

Third, additional widgets. One of the advantages of dashboards in comparison to standard web maps is that they allow to bring elements of analysis to a more finished product. Think about what you want your users to be able to analyse, why, and how that will modify the main map. This is the icing on the cake!

Submit

Once completed, you will submit a report through Turnitin that includes the following:

- A link to the published dashboard, which needs to be reachable online
- About 250 words for the overall idea of the dashboard. What do you want to communicate? What is the story you want to tell?
- About 250 words for the data used. Which datasets are you using? Why? What new information do they bring and how they complement each other?
- About 250 words to describe your design choices in the basemap and other layers presented (e.g. choropleths).

- About 250 words to describe your design choices around interactivity, including both cartographic elements (e.g. zooming, panning) as well as additional interactivity built around components such as widgets.

The assignment will be evaluated based on:

1. *Overall design of the experience.* It is very important you think through every step of preparing this assignment as if it was part of something bigger towards which it contributes. Because that is exactly what it is. Everything should have a reason to be there, and every aspect of the dashboard should be connected to each other following a common thread. And, of course, make this connection and holistic approach come alive in your report.
2. *(Base)map design.* Critically introduce every aspect you have thought about when designing the maps, and explicitly connect it to the overal aim of the dashboard. Be clear in your descriptions and critical in how you engage every design choice.
3. *Interactivity design.* Your dashboard should use interactivity when necessary to deliver a more compelling and fuller experience that better gets your message across. Be sure to clearly lay out in your report which elements are used and why.
4. *Narrative around the description of the process.* Finally, the final mark will also take into account not only how good your dashboard is, but how well you are able to introduce it. Start with the key goals, and then unpack every element in an integrated and compelling way.

How is this assignment useful?

This assignment combines several elements that will help you improve critical aspects of web mapping:

- *Design*: this is not about making maps, this is about making good maps. And behind every good map there is a set of conscious choices that you will have to think throug to be successful (what map? what data? how to present the data? etc.).
- *Technology*: at the end of the day, building good web maps requires familiarity with the state-of-the-art in terms of web mapping tools. In this assignment, you will need to demonstrate your mastering of some of the key tools that are leading both industry and academia.
- *Presentation*: in many real-world contexts, your work is as good as it can come across to the audience it is intended to. This means that it is vital to be able to communicate not only what you are doing but why and on what building blocks it is based on.

Marking Criteria

This course follows the standard marking criteria (the general ones and those relating to GIS assignments in particular) set by the School of Environmental Sciences. Please make sure to check the student handbook and familiarise with them. In addition to these generic criteria, the following specific criteria will be used in cases where computer code is part of the work being assessed:

- 0-15: the code does not run and there is no documentation to follow it.
- 16-39: the code does not run, or runs but it does not produce the expected outcome. There is some documentation explaining its logic.
- 40-49: the code runs and produces the expected output. There is some documentation explaining its logic.
- 50-59: the code runs and produces the expected output. There is extensive documentation explaining its logic.
- 60-69: the code runs and produces the expected output. There is extensive documentation, properly formatted, explaining its logic.
- 70-79: all as above, plus the code design includes clear evidence of skills presented in advanced sections of the course (e.g. custom methods, list comprehensions, etc.).
- 80-100: all as above, plus the code contains novel contributions that extend/improve the functionality the student was provided with (e.g. algorithm optimizations, novel methods to perform the task, etc.).

1 Introduction

Dani Arribas-Bel

Lecture: Introduction to the module

Lab: Powerful examples & Discussion about interactive map

1.1 Lecture

Slides can be downloaded "here"

1.2 Lab: Powerful examples

This lab has two main components:

1. The first one will require you to find a partner and work together with her/him
2. And the second one will involve group discussion.

1.2.1 Paired activity

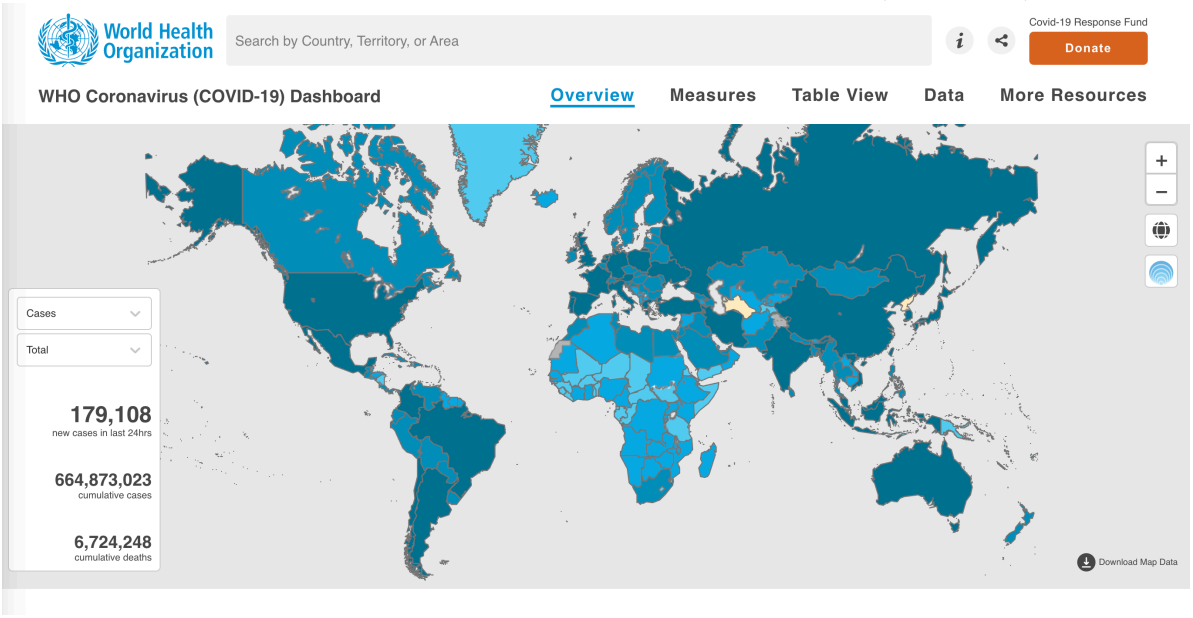
In pairs, find three examples where web maps are used to communicate an idea. Complete the following sheet for each example:

- **Substantive**
  - `Title`: Title of the map/project
  - `Author`: Who is behind the project?
  - `Big idea`: a "one-liner" on what the project tries to accomplish –
  - `Message`: what does the map try to get accross
- **Technical**
  - `URL`:
  - `Interactivity`: does the map let you interact with it in any way? Yes/No

- `Zoomable`: can you explore the map at different scales? Yes/No
- `Tooltips`:
- `Basemap`: Is there an underlying map providing geographical context? Yes/No. If so, who is it provided by?
- `Technology`: can you guess what technology does this map rely on?

Post each sheet as a separate item on the Teams channel for Lab No.1

As an example, below is the sheet for the project "WHO Coronavirus (COVID-19) Dashboard"



- **Substantive**
  - `Title`: WHO Coronavirus (COVID-19) Dashboard
  - `Author`: World Health Organization
  - `Big idea`: Shows confirmed COVID-19 cases and deaths by country to date
  - `Message`: The project displays a map of the world where COVID-19 cases are shown by country. This element is used to show which countries have had more cases (large trends). A drop down button allows us to visualise the map by a) Total per 100,000 population b) % change in the last 7 days c) newly reported in the last 7 days d) newly reported in the last 24 hours.
- **Technical**
  - `URL`: https://covid19.who.int/
  - `Interactivity`: Yes

- `Zoomable`: Yes
- `Tooltips`: Yes
- `Basemap`: No
- `Technology`: Unknown

Here are a couple of other COVID-19 examples of web-maps that where basemaps and technology is easier to spot.

- "London School of Hygiene & Tropical Medicine - COVID-19 tracker"
- "Tracking Coronavirus in the United Kingdom: Latest Map and Case Count"

1.2.2 Class discussion

We will select a few examples posted and collectively discuss (some of) the following questions:

1. What makes them powerful, what "speaks" to us?
2. What could be improved, what is counter-intuitive?
3. What design elements do they rely on?
4. What technology do they use?

1.3 References

- For an excellent coverage of "visualisation literacy", Chapter 11 of Andy Kirk's "Data Visualisation" is a great start. Lab: Getting up to speed for web mapping
- A comprehensive overview of computational notebooks and how they relate to modern scientific work is available on Ch.1 of the GDS book.
- A recent overview of notebooks in Geography is available in Boeing & Arribas-Bel (2021)

2 Web architecture

Dani Arribas-Bel

Lecture: The Web's architecture and Economy

Lab: What do APIs actually do? (non-spatial APIs)

2.1 Lecture

Slides can be downloaded here

2.2 Lab: What do APIs actually do?

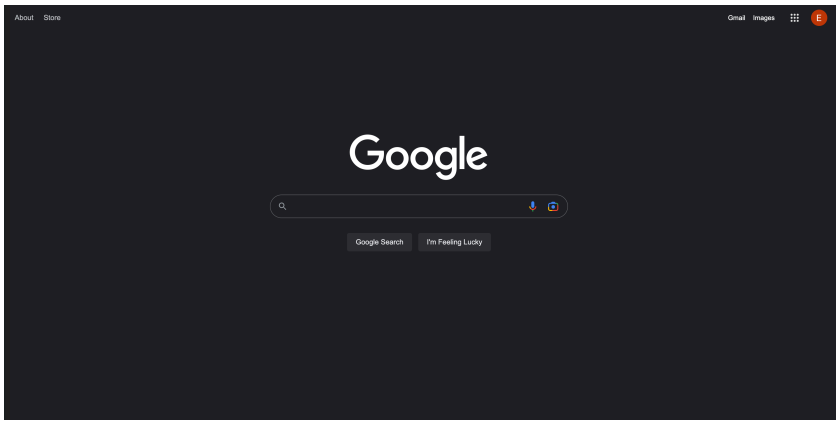
In this lab, we will unpack how Application Programming Interfaces ("APIs") work and we cover the basics of accessing an API using R. Instead of downloading a data set, APIs allow programmers, statisticians (or students) to request data directly from a server to a local machine. When you work with web APIs, two different computers — a client and server — will interact with each other to request and provide data, respectively.

2.2.1 RESTful Web APIs are all around you.

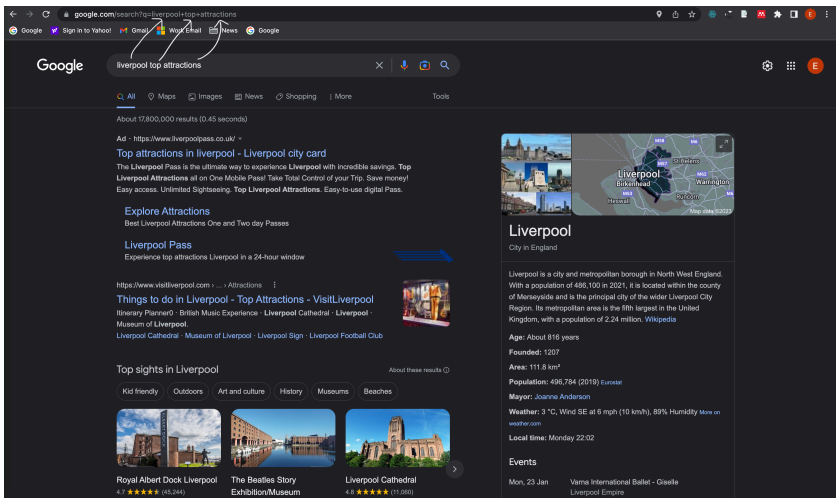
Web APIs

- Allow you query a remote database over the internet
- Take on a variety of formats
- Adhere to a particular style known as Representation State Transfer or REST (in most cases)
- RESTful APIs are convenient because we use them to query database using URLs

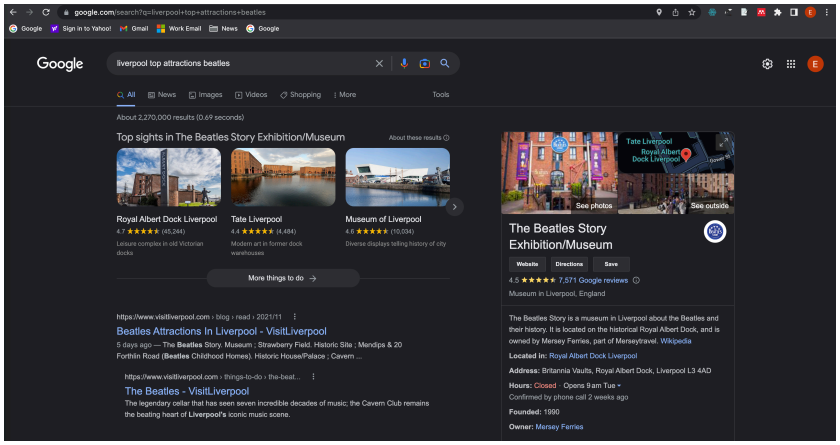
Consider a simple Google search:



Ever wonder what all that extra stuff in the address bar was all about? In this case, the full address is Google's way of sending a query to its databases requesting information related to the search term *liverpool top attractions*.



In fact, it looks like Google makes its query by taking the search terms, separating each of them with a +, and appending them to the link https://www.google.com/#q=. Therefore, we should be able to actually change our Google search by adding some terms to the URL:



Learning how to use RESTful APIs is all about learning how to format these URLs so that you can get the response you want.

2.2.2 Group activity

Get into groups of 5 or 6 students. Using your friend the internet, look up answers to the following questions. Each group will be assigned one question and asked to present their findings in 5 min to discuss with the entire class.

1. What is a `URL` and how can it help us query data? What is a response status and what are the possible categories?
2. What is a `GET` request? How does a `GET` request work?
3. What are API keys and how do you obtain them? What kinds of restrictions to they impose on users? Find an example of an API key, what does it look like?
4. (For 2 groups) More and more APIs pop up every day. Do a bit of quick research and find 2 different examples of APIs that you would be interested in using. 2 groups, 2 or 3 APIs each.

2.2.3 API R libraries

There are two ways to collect data through APIs in R:

Plug-n-play packages. Many common APIs are available through user-written R Packages. These packages offer functions that conveniently "wrap" API queries and format the response. These packages are usually much more convenient than writing our own query, so it is worth searching for a package that works with the API we need.

Writing our own API request. If no wrapper function is available, we have to write our own API request and format the response ourselves using R. This is tricky, but definitely doable.

2.2.3.1 `tidycensus` pair activity

Some R packages "wrap" API queries and format the response. Lucky us! In pairs, let's have a look at `tidycensus`. You can also have a look at the different APIs available from the United States Census Bureau.

```
tidycensus is

- R package first released in mid-2017

- Allows R users to obtain decennial Census and ACS data pre-formatted for
use with with tidyverse tools (dplyr, ggplot2, etc.)

- Optionally returns geographic data as simple feature geometry for common
Census geographies
```

Create a new `R-markdown` and save it to something you'll remember, like `web_mapping_lab_02.Rmd`. To get started working, load the package along with the `tidyverse` and `plyr` packages, and set you Census API key. A key can be obtained from http://api.census.gov/data/key_signup.html.

```
library(plyr)
library(tidycensus)
library(tidyverse)

#census_api_key("ADD IT HERE") #replace this with your key
```

- Variables in tidycensus are identified by their Census ID, e.g. B19013_001
- Entire tables of variables can be requested with the table argument, e.g. table = "B19001"
- Users can request multiple variables at a time, and set custom names with a named vector

Searching for variables Getting variables from the US American Community Survey (ACS) 5-Year Data (2016-2020) requires knowing the variable ID - and there are thousands of these IDs across the different files. To rapidly search for variables, use the load_variables() function. The function takes two required arguments: the year of the Census or endyear of the ACS sample, and the dataset name, which varies in availability by year. For the ACS, use either "acs1" or "acs5" for the ACS detailed tables, and append /profile for the Data Profile and /subject for the Subject Tables. To browse these variables, assign the result of this function to

a variable and use the View function in RStudio. An optional argument cache = TRUE will cache the dataset on your computer for future use.

```
view_vars <- load_variables(2020, "acs5", cache = TRUE)

view(view_vars)
```

EXERCISE - In your pairs explore some of the different variables available in the 5-Year ACS (2016-2020). Make a note of 3 variables you would be interested in exploring. The ACS2 variables page might also help.

```
income <- get_acs(geography = "state", table = "B19001") #getting income data by state
income
```

```
# A tibble: 884 x 5
   GEOID NAME    variable   estimate   moe
   <chr> <chr>   <chr>         <dbl> <dbl>
 1 01    Alabama B19001_001  1888504  5749
 2 01    Alabama B19001_002   153635  2979
 3 01    Alabama B19001_003   105415  2397
 4 01    Alabama B19001_004   106327  2522
 5 01    Alabama B19001_005   100073  2674
 6 01    Alabama B19001_006   100569  3023
 7 01    Alabama B19001_007    96815  2745
 8 01    Alabama B19001_008    87120  2491
 9 01    Alabama B19001_009    86181  2124
10 01    Alabama B19001_010    75721  2512
# ... with 874 more rows
```

EXERCISE - 1) What is a tibble? 2) Discuss the format of the data obtained with your partner and then use the function `get_acs` to explore the 3 variables you discussed in the previous exercise.

You can also get "wide" census data:

```
inc_wide <- get_acs(geography = "state", table = "B19001", output = "wide")
inc_wide
```

```
# A tibble: 52 x 36
   GEOID NAME    B1900~1 B1900~2 B1900~3 B1900~4 B1900~5 B1900~6 B1900~7 B1900~8
   <chr> <chr>     <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
```

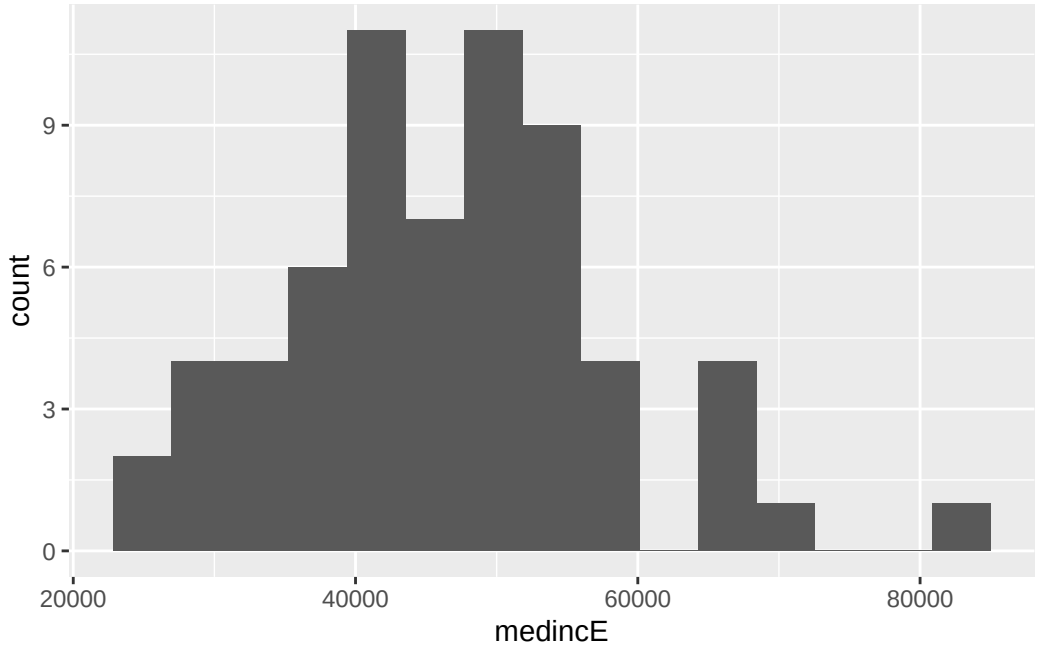
```
 1 42    Pennsy~  5.11e6    8064  296733    3893  206216    3771  223380    3598
 2 06    Califo~  1.31e7   18542  614887    6699  507398    5286  435382    5251
 3 54    West V~  7.34e5    2810   62341    2318   43003    1497   45613    1648
 4 49    Utah     1.00e6    2384   36211    1536   27395    1378   28460    1507
 5 36    New Yo~  7.42e6   12559  471680    6161  340614    4703  303901    4201
 6 11    Distri~  2.88e5    1319   24083    1442   11315     809    8300     842
 7 02    Alaska   2.55e5    1326    9818     613    7476     651    8007     633
 8 12    Florida  7.93e6   23200  494959    6755  329848    4816  354967    5030
 9 45    South ~  1.96e6    5748  144667    3397   93868    2582   94132    2606
10 38    North ~  3.21e5    1737   18120     846   12664     795   12611     874
# ... with 42 more rows, 26 more variables: B19001_005E <dbl>,
#   B19001_005M <dbl>, B19001_006E <dbl>, B19001_006M <dbl>, B19001_007E <dbl>,
#   B19001_007M <dbl>, B19001_008E <dbl>, B19001_008M <dbl>, B19001_009E <dbl>,
#   B19001_009M <dbl>, B19001_010E <dbl>, B19001_010M <dbl>, B19001_011E <dbl>,
#   B19001_011M <dbl>, B19001_012E <dbl>, B19001_012M <dbl>, B19001_013E <dbl>,
#   B19001_013M <dbl>, B19001_014E <dbl>, B19001_014M <dbl>, B19001_015E <dbl>,
#   B19001_015M <dbl>, B16001_016E <dbl>, B19001_016M <dbl>, ...
```

Let's make our query a bit more precise. We are going to query data on median household income and median age by county in the state of New York from the 2016-2020 ACS.

```
ga_wide <- get_acs(
  geography = "county",
  state = "Louisiana",
  variables = c(medinc = "B19013_001",
                medage = "B01002_001"),
  output = "wide",
  year = 2020
)
```

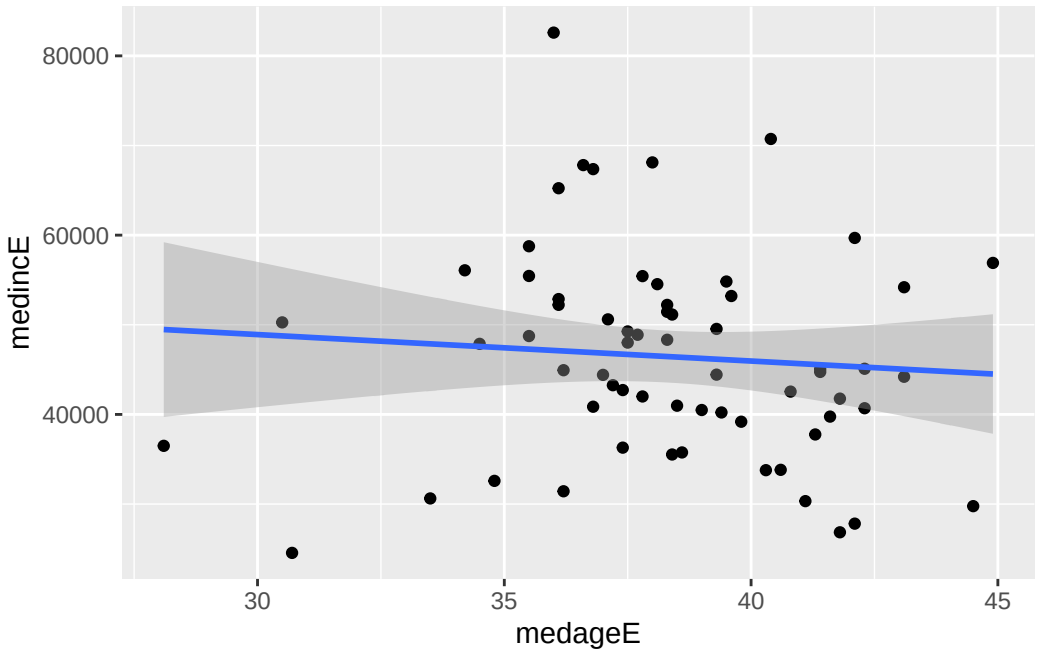
Let's plot one of our variables. By default, ggplot organizes the data into 30 bins; this option can be changed with the bins parameter.

```
ggplot(ga_wide, aes(x = medincE)) +
  geom_histogram(bins = 15) #argument bins = 15 in our call to geom_histogram()
```



We can also easily explort correlations between variables. The `geom_point()` function, which plots points on a chart relative to X and Y values for observations in a dataset. This requires specification of two columns in the call to `aes()`.

```
ggplot(ga_wide, aes(x = medageE, y = medincE)) +
  geom_point() +
  geom_smooth(method = "lm")
```



EXERCISE - In your pairs, modify the state, vairables and year parametres in your `get_acs` function and produce some other simple scatter plots (cloud of points) that suggest correlations between your variables of interest.

You can also directly map data you have queried in `tidycensus`. We will look at this in future sessions. For a complete overview of `tidycensus` please see Analyzing US Census Data: Methods, Maps, and Models in R.

2.2.3.2 Your own API request demo

The R libraries that are often used for APIs are `httr` and `jsonlite`. They serve different roles in our introduction of APIs, but both are essential.

`JSON` stands for JavaScript Object Notation. While JavaScript is another programming language. JSON is useful because it is easily readable by a computer, and for this reason, it has become the primary way that data is transported through APIs. Most APIs will send their responses in JSON format. Using the `jsonlite` package, you can extract and format data into an R dataframe. `JSON` is a structure formatted with a key (for example, a variable name `id`) and a value (`BikePoints_308`). We used the function `fromJSON` to transform the API request content into a useable dataframe.

We will request the locations of all the hire bike stations in London from the Transport for London API. We use the `GET` function from `httr` package.The `GET()` function requires a URL, which specifies the server's address to which the request needs to be sent.

```
library(httr)
library(jsonlite)

#key <- "YOURKEY HERE"
request <- GET("https://api.tfl.gov.uk/BikePoint/") # Here we request all the bike docking
```

```
request # Examine output
```

```
Response [https://api.tfl.gov.uk/BikePoint/]
  Date: 2023-03-13 22:46
  Status: 200
  Content-Type: application/json; charset=utf-8
  Size: 2.17 MB
```

```
request$status_code # The response status is 200 for a successful request
```

```
[1] 200
```

Most GET request URLs for API querying have three or four components:

1. Authentication Key/Token: A user-specific character string appended to a base URL telling the server who is making the query; allows servers to efficiently manage database access.
2. Base URL: A link stub that will be at the beginning of all calls to a given API; points the server to the location of an entire database.
3. Search Parameters: A character string appended to a base URL that tells the server what to extract from the database; basically a series of filters used to point to specific parts of a database.
4. Response Format: A character string indicating how the response should be formatted; usually one of .csv, .json, or .xml.

```
bikepoints <- jsonlite::fromJSON(content(request, "text")) # extract the dataframe
names(bikepoints) # Print the column names
```

```
 [1] "$type"                "id"                   "url"
 [4] "commonName"           "placeType"            "additionalProperties"
 [7] "children"             "childrenUrls"         "lat"
[10] "lon"
```

```
bikepoints$`Station ID` = as.numeric(substr(bikepoints$id, nchar("BikePoints_")+1, nchar(b
```

After Block 3 Data architectures we will have revised spatial data forms and you will easily be able to map data that you have obtained through this API.

```
## Create an sf object from longitude latitude
library(dplyr)
library(sf)
library(tmap)
# create a sf object
stations_df <- bikepoints %>%
  sf::st_as_sf(coords = c(10,9))  %>%  # create pts from coordinates
  st_set_crs(4326) %>%  # set the original CRS
  relocate(`Station ID`) # set ID as the first column of the dataframe

# plot bikepoints on a background map for more context
tmap_mode("view")
tm_basemap() +
  tm_shape(stations_df) +
  tm_symbols(id = "commonName", col = "red", scale = .5)
```

2.3 Group activity answers

1. Uniform Resource Location (`URL`) is a string of characters that, when interpreted via the Hypertext Transfer Protocol (`HTTP`). URLs point to a data resource, notably files written in Hypertext Markup Language (`HTML`) or a subset of a database
    - 1xx informational response - the request was received, continuing process
    - 2xx successful - the request was successfully received, understood, and accepted
    - 3xx redirection - further action needs to be taken in order to complete the request
    - 4xx client error - the request contains bad syntax or cannot be fulfilled
    - 5xx server error - the server failed to fulfil an apparently valid request

2. `GET` requests a representation of a data resource corresponding to a particular `URL`. The process of executing the `GET` method is often referred to as a `GET request` and is the main method used for querying RESTful databases. `HEAD`, `POST`, `PUT`, `DELETE`: other common methods, though mostly never used for database querying.

   Surfing the web is basically equivalent to sending a bunch of `GET` requests to different servers and asking for different files written in `HTML`. Suppose, for instance, I wanted to look something up on Wikipedia. Your first step would be to open your web browser and type in http://www.wikipedia.org. Once you hit return, you would see the page below. Several different processes occured, however, between you hitting "return" and the page finally being rendered:

   1. The web browser took the entered character string, used the command-line tool "Curl" to write a properly formatted `HTTP GET` request, and submitted it to the server that hosts the Wikipedia homepage.
   2. After receiving this request, the server sent an `HTTP` response, from which `Curl` extracted the HTML code for the page (partially shown below).
   3. The raw `HTML` code was parsed and then executed by the web browser, rendering the page as seen in the window.
   4. Most APIs requires a key or other user credentials before you can query their database. Getting credentialised with a API requires that you register with the organization. Once you have successfully registered, you will be assigned one or more keys, tokens, or other credentials that must be supplied to the server as part of any API call you make. To make sure users are not abusing their data access privileges (e.g., by making many rapid queries), each set of keys will be given rate limits governing the total number of calls that can be made over certain intervals of time.

3. Most APIs requires a key before you can query their database. This usually requires you to register with the organization. Most APIs are set up for developers, so you will likely be asked to register an "application." All this really entails is coming up with a name for your app/bot/project and providing your real name, organization, and email. Note that some more popular APIs (e.g., Twitter, Facebook) will require additional information, such as a web address or mobile number. Once you have registered, you will be assigned one or more keys, tokens, or other credentials that must be supplied to the server as part of any API call you make. Most API keys limits he total number of calls that can be made over certain intervals of time. This is so users do not busing their data access privileges.

2.4 References

- Brief History of the Internet, by the Internet Society, is a handy (and free!) introduction to how it all came to be.
- Haklay, M., Singleton, A., Parker, C. 2008. "Web Mapping 2.0: The Neogeography of the GeoWeb". Geography Compass, 2(6):2011–2039
- A blog post from Joe Morrison commenting on the recent change of licensing for some of the core software from Mapbox
- Terman, R., 2020. Computational Tools for Social Science
- Walker, K. Analyzing US Census Data: Methods, Maps, and Models in R.

3 Data architectures

Elisabetta Pietrostefani

Lecture: Data architectures & formats

Lab: Creating, manipulating, and integrating web geo-spatial data

3.1 Lecture

Slides can be downloaded here

3.2 Lab: Creating, manipulating, and integrating web geo-spatial data

In this lab, we will explore and familiarise with some of the most common data formats for web mapping: GeoJSON and Mbtiles. To follow this session, you will need to be able to access the following:

- The internet
- QGIS. Any version should work in this context, but if you are installing it on your computer, QGIS 3.22 is recommended
- The R libraries listed in the computational environment setup of the course.

3.2.1 GeoJSON

To get familiar with the format, we will start by creating a GeoJSON file from scratch. Head over to the following website:

https://geojson.io/

In there, we will create together a small example to better understand the building blocks of this file format.

We will pay special attention to the following aspects:

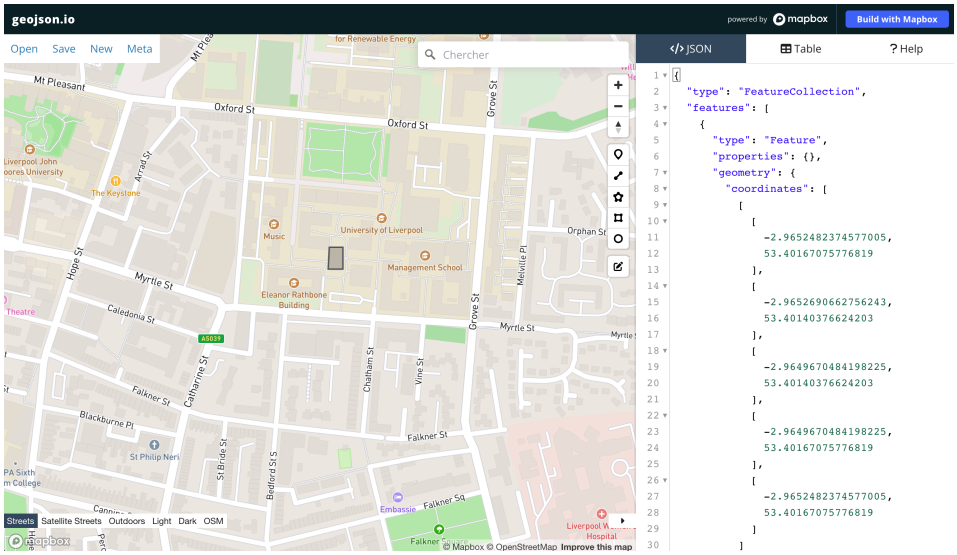


Figure 3.1: geojson.io

- Readability
- Coordinate system
- Ability to add non-spatial information attached to each record
- How to save it as a file

EXERCISE

Create a GeoJSON file for the following data and save them to separate files:

1. Your five favourite spots in Liverpool
2. A polygon of what you consider to be the boundary of the neighbourhood where you live and the city centre of Liverpool. Name each.
3. A route that captures one of your favorite walks around the Liverpool region

If you are comfortable, upload the files to Microsoft Teams to share them with peers.

3.2.2 GeoJSON in R

With the files from the exercise at hand, we will then learn how to open them in `R-markdown`. Create a new `R-markdown` and save it to something you'll remember, like `web_mapping_lab_03.Rmd`.

Then let's start by calling the libraries `sf` and `geojsonsf` :

```
# Simple features, a standardized way to encode spatial vector data
library(sf)
# Converts Between GeoJSON and simple feature objects
library(geojsonsf)
```

Now, place the .geojson files you have created in the same folder where you are storing the `R-markdown`, or somewhere reachable. For this example, we will assume that the file is called `map.geojson` and it is stored in the `data` folder, accessible from the same location where the notebook is. We can read the file as:

```
liverpool <- geojson_sf("data/map.geojson")
```

We can inspect the file to see what it contains:

```
head(liverpool)
```

```
Simple feature collection with 4 features and 0 fields
Geometry type: GEOMETRY
Dimension:     XY
Bounding box:  xmin: -2.977367 ymin: 53.39987 xmax: -2.954183 ymax: 53.40753
Geodetic CRS:  WGS 84
                        geometry
1 POLYGON ((-2.965248 53.4016...
2 LINESTRING (-2.975764 53.40...
3     POINT (-2.977367 53.40753)
4 POLYGON ((-2.958036 53.4009...
```

If you are familiar with `sf` objects, this is exactly it, read straight from a GeoJSON file (if you need a refresher, you can check out introduction to `sf`) in the Spatial Data Science book.

A point is single point geometry: `POINT (5 2)`

A line string is a sequence of points with a straight line connecting the points: `LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)`

A polygon is a sequence of points that form a closed ring without intersection. Closed means that the first and the last point of a polygon have the same coordinates: `POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))`

Let's quickly plot the `sf` object to visualise it in R.

```
# Provides functions to very quickly and conveniently create interactive visualisations of
library(mapview)
```

```
mapview(liverpool)
```

Once read, the geojson behaves exactly like any `sf` objects, we can therefore operate on it and tap into the functionality from `sf`. For example, we can inspect the Coordinate Reference System (CRS) in which it is expressed:

```
st_crs(liverpool)
```

```
Coordinate Reference System:
  User input: 4326
  wkt:
GEOGCS["WGS 84",
      DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
          AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
      PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
      UNIT["degree",0.0174532925199433,
        AUTHORITY["EPSG","9122"]],
      AXIS["Latitude",NORTH],
      AXIS["Longitude",EAST],
    AUTHORITY["EPSG","4326"]]
```

Using some of `sf`'s functionality. We can reproject it to express it in metres:

```
# Transform to British National Grid
liverpool_bng <- st_transform(liverpool, st_crs(27700))
# To check the new projection
st_crs(liverpool_bng)
```

```
Coordinate Reference System:
  User input: EPSG:27700
  wkt:
PROJCRS["OSGB36 / British National Grid",
    BASEGEOGCRS["OSGB36",
        DATUM["Ordnance Survey of Great Britain 1936",
            ELLIPSOID["Airy 1830",6377563.396,299.3249646,
                LENGTHUNIT["metre",1]]],
        PRIMEM["Greenwich",0,
```

```
        ANGLEUNIT["degree",0.0174532925199433]],
    ID["EPSG",4277]],
CONVERSION["British National Grid",
    METHOD["Transverse Mercator",
        ID["EPSG",9807]],
    PARAMETER["Latitude of natural origin",49,
        ANGLEUNIT["degree",0.0174532925199433],
        ID["EPSG",8801]],
    PARAMETER["Longitude of natural origin",-2,
        ANGLEUNIT["degree",0.0174532925199433],
        ID["EPSG",8802]],
    PARAMETER["Scale factor at natural origin",0.9996012717,
        SCALEUNIT["unity",1],
        ID["EPSG",8805]],
    PARAMETER["False easting",400000,
        LENGTHUNIT["metre",1],
        ID["EPSG",8806]],
    PARAMETER["False northing",-100000,
        LENGTHUNIT["metre",1],
        ID["EPSG",8807]]],
CS[Cartesian,2],
    AXIS["(E)",east,
        ORDER[1],
        LENGTHUNIT["metre",1]],
    AXIS["(N)",north,
        ORDER[2],
        LENGTHUNIT["metre",1]],
USAGE[
    SCOPE["Engineering survey, topographic mapping."],
    AREA["United Kingdom (UK) - offshore to boundary of UKCS within 49°45'N to 61°N and 9
    BBOX[49.75,-9,61.01,2.01]],
ID["EPSG",27700]]
```

When we inspected our geojson with `head(liverpool)` we noted that the spatial data is stored in the following format: `POINT (-2.977367 53.40753)`. This is called "well known text" (`wkt`) and is a representation that spatial databases like PostGIS use as well. Another way to store spatial data as text for storage or transfer, less (human) readable but more efficient is the "well known blurb" (`wkb`). In R, you can transform `wkt` data into `wkb` data using the library `wellknown`and the function `wkt_wkb`.

```
library(wellknown) # Convert Between 'WKT' and 'GeoJSON'
```

```r
# Load the WKT representation of the point and convert the WKT representation into WKB for
wkt <- wkt_wkb("POINT (-2.977367 53.40753)")

# Print the WKB data
wkt
```

```
 [1] 01 01 00 00 00 5e 84 29 ca a5 d1 07 c0 c7 11 6b f1 29 b4 4a 40
```

Another benefit of reading data in R is we can use its analytical capabilities. For example, we can calculate the length of a line our data frame:

```r
# Extract the second row, which is a line, and reconvert to sf object
liverpool_walk <- st_sf(liverpool_bng[2,])

# Calculate the length of the linestring
st_length(liverpool_walk)
```

```
800.678 [m]
```

Given the the line is expressed in metres (check out EPSG:27700), we can conclude the line spans about 800.678 metres,

EXERCISE

- Read the `GeoJSON` created for your favorite walks in Liverpool and calculate their length
- **Pro**: explore the `R documentation` and try to extract the area for the polygon covering your neighbourhood

Once you are happy with the data as we will hypothetically need it, you can write it out to any other file format supported in `sf`. For example, we can create a Geopackge file with the same information. For this, we can use the function `st_write`. The file name is taken as the data source name. The default for the layer name is the basename (filename without path) of the the data source name. See an example below:

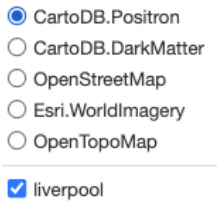
```r
## Writing layer `liverpool_bng' to data source `liverpool_bng.gpkg' using driver `Geopack
st_write(liverpool_bng, dsn = "data/liverpool_bng.gpkg", layer = "data/liverpool_bng.gpkg"
```

```
Deleting source `data/liverpool_bng.gpkg' using driver `GPKG'
Writing layer `data/liverpool_bng.gpkg' to data source
  `data/liverpool_bng.gpkg' using driver `GPKG'
Writing 4 features with 0 fields and geometry type Unknown (any).
```

R's `sf` cheatsheet is a good reference for manipulation operations/spatial predicates with simple features.

3.2.3 Tilesets and `Mbtiles`

In this section we will dive into the concept of tiles to understand why they have been so transformative in the world of web mapping. We have already seen several tilesets. If you scroll back up where we used the function `mapview`, you will see that is it plotting out points, polygons and linestrings on different tileset options integrated in the library.



We will learn how to prepare a map that is styled in QGIS and then saved as either an .mbtiles file of a structured folder with tiles that allows to serve it over the web in efficient ways. Finally we will explore the tileset built using the JavaScript library Leaflet.js.

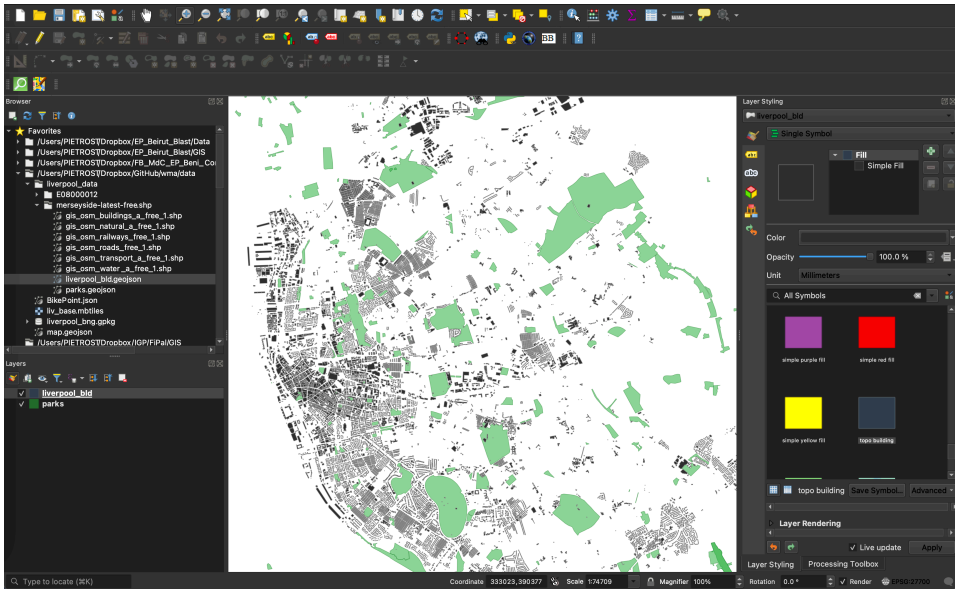
Before we get started, let's get all the required pieces together:

- Fire up QGIS 3
- Download the data here

Build your basemap

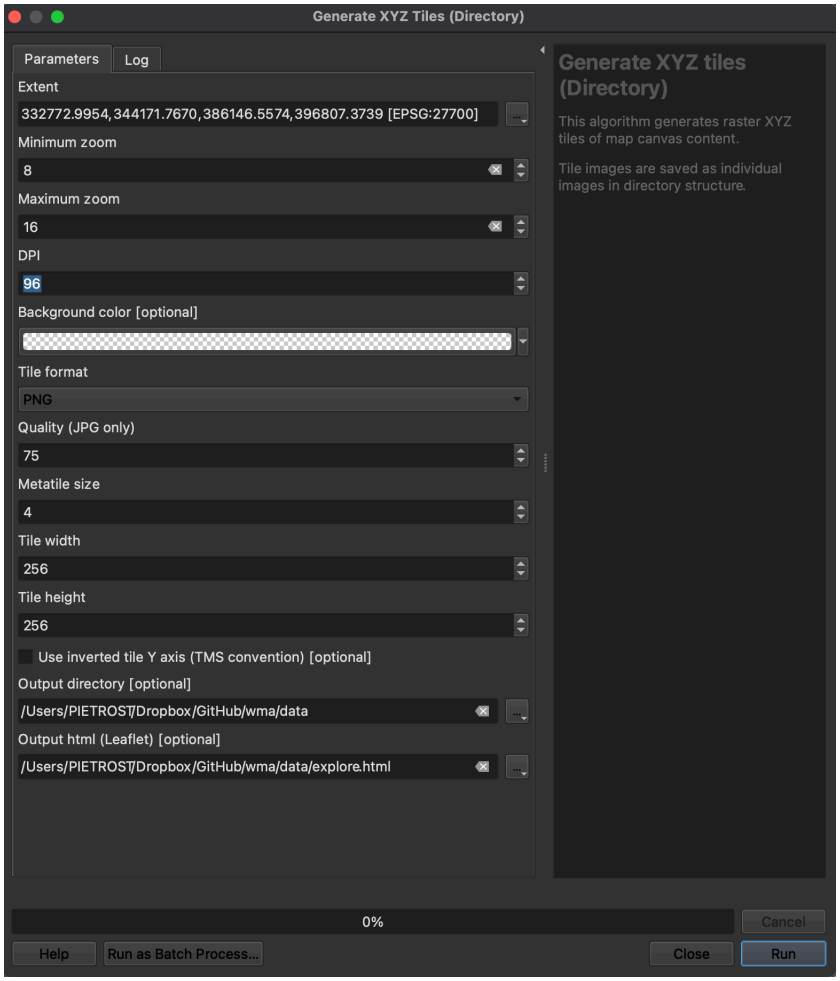
Basemaps are maps that provide context to more specific spatial data you might want to present. For example, if you have a set of points that represent events in space, it might be hard to understand their distribution unless you put them in the context of a more complete geography. A basemap is a quick solution in this case.

Explore the layers provided in the GeoData Pack and select those you want to use for your basemap. Once ready, go ahead and add them as layers in QGIS. Tweak colors, transparencies, linewidths, etc. until you get a map you are happy with.



Create a tileset for your map

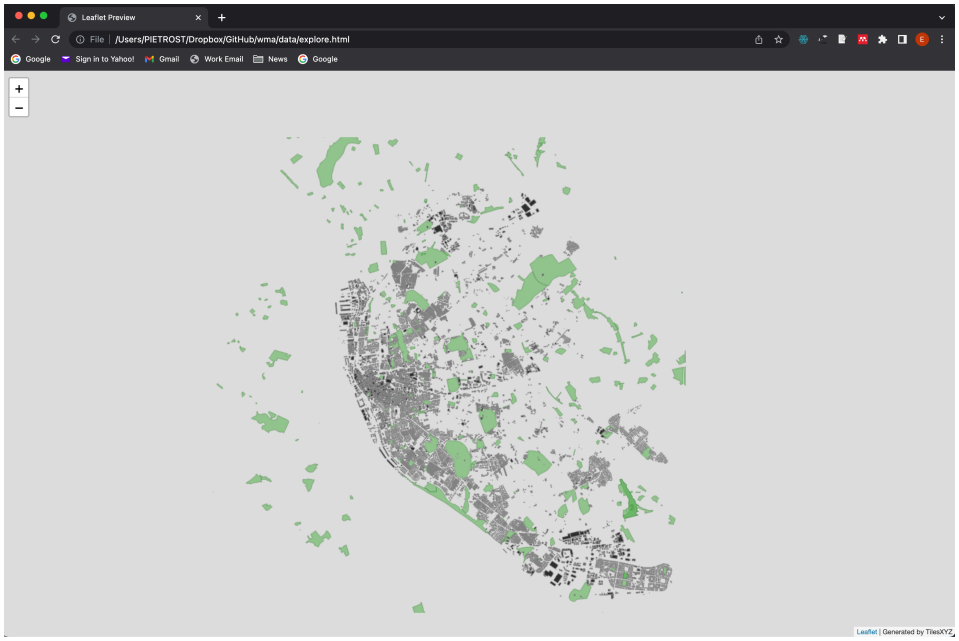
Once ready to build the basemap from your created map, head over to `Processing Toolbox` and select the `Raster tools --> Generate XYZ Tiles (Directory)`. You can start with the Directory option. Pick parameters and, when everything is ready, hit Run. Depending on your settings, this will take some time, be patient.



When finished, QGIS will have created a folder with a particular structure, that contains *all* the tiles required to serve your basemap. You can peak into them to find they are really just images of different parts of your map at different zoom levels.

Explore your basemap with Leaflet.js

If you store your basemap in a folder, QGIS will also generate for you a HTML file with a bit of JavaScript code that will allow you to explore the tileset in a browser. Play with it a little bit and familiarise with the look and feel of it.



If you feel adventurous, you can also peak into the code that makes the web map possible. To do that, you will need to either open the HTML file on a text editor, or inspect the source code from the browser (in Chrome, for example, this can be accessed through `Right Click --> Inspect`.

Create a `.mbtiles` file for easy transport

Finally, you can recreate the process above but in this case choosing the MBTiles instead of the Directory option. This will make QGIS generate the same tileset but, instead of storing it directly on a folder, it will save it as a SQLite database in with the `.mbtiles` format. This is easier to move from one environment to another and is also supported by most web mapping platforms, such as Mapbox.

`.mbtiles` in R

For this next section, you will need to register for a MapBox account. here.

Following from the previous exercise, save a file of Liverpool buildings as a geojson and call it `buildings_liverpool.geojson`.

```
# R Interface to 'Mapbox' Web Services
library(mapboxapi)
```

```
Usage of the Mapbox APIs is governed by the Mapbox Terms of Service.
Please visit https://www.mapbox.com/legal/tos/ for more information.
```

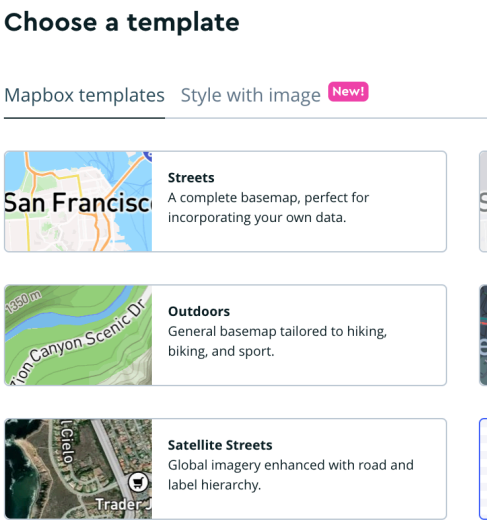
```
# Mapdeck is a combination of Mapbox and Deck.gl. Deck.gl is one of the most user-friendly
library(mapdeck)
#Tools for Working with URLs and HTTP
library(httr)
```

You can use tippecanoe to make a dynamic .mbtiles files in R, just like the `XYZ function` in QGIS. This is useful to visualize large data appropriately at any zoom level. `sf` objects can also be used as input! This requires you to install tippecanoe on your machine separately first. See instructions here.

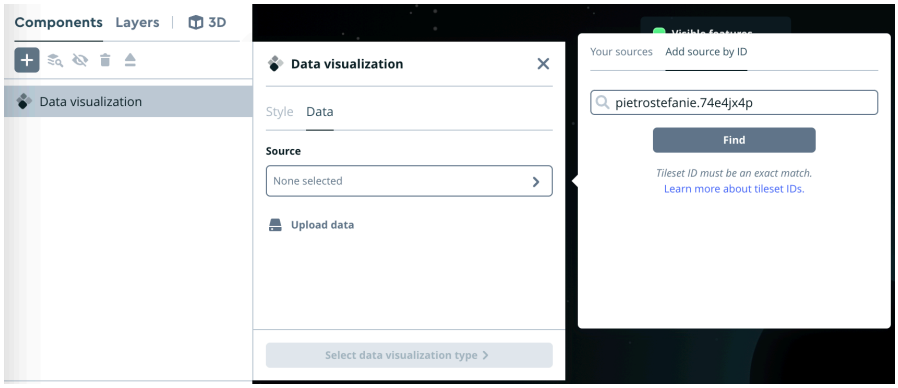
```
# Using tippecanoe to create .mbtiles of building footprint liverpool
tippecanoe(input = "data/buildings_liverpool.geojson",
           output = "data/buildings_liverpool.mbtiles",
           layer_name = "buildings_liverpool")
```

You can then upload the generated tilesetwith the `upload_tiles()` function to your Mapbox account (requires a Mapbox secret access token to be set as an environment variable). Or you can upload this manually.

- Head over to Mapbox Studio



- Start a `New Style` and chose a template (Monochrome, blank etc.)
- Upload the data and style it



It should look like a nicer version of this:

When you've styled it, bring it back into your R project

```
# Define your mapbox token
#my_token <- "PLACE YOUR MAPBOX TOKEN HERE and UNCOMMENT"

# Into R with mapdeck by referencing the style ID.
mapdeck(token = my_token,
        # Replace the style with the one you've created
        style = "mapbox://styles/pietrostefanie/cle4dgs2o002d01tft37b5ndg",
        zoom = 10,
        location = c(-2.973286, 53.406872))
```

3.3 References

- Pebesma, E. & Bivand, R. (2022) Spatial Data Science with applications in R
- Chapter 3 of the GDS book (in progress) covers traditional and more modern approaches to represent Geography as data.
- Kitchin, R. (2014). The data revolution: Big data, open data, data infrastructures and their consequences. Sage.
- Maptiler.com documents on map tiles and map vector tiles.

4 APIs

Elisabetta Pietrostefani

Lecture: APIs

Lab: Acquiring data from the web.

4.1 Lecture

Slides can be downloaded here

4.2 Lab: Acquiring data from the web.

In this lab, we will interact with a few APIs to get a feel for how they work and how you can make the most of them when trying to access data on the web. To follow this session, you will need to be able to access the following:

- The internet
- A Mapbox API token, which you can access through your Mapbox account. If you haven't signed up, do this before class here.
- The R libraries listed in the computational environment setup of the course.

4.2.1 Basemap API

This section will cover the access of basemaps served as tilesets through the standard XYZ protocol. For this, we will use the library `leaflet`. It is an open-source Javascript library and a popular option for creating interactive mobile-friendly maps. We will use it first as end-users, and then we will peak a bit into its guts to get a better understanding of its inner workings.

`Leaflet` provider list - The `leaflet` packages comes with 100+ provider tiles - The names of these tiles are stores in a list named `providers`

As a convenience, leaflet also provides a named list of all the third-party tile providers that are supported by the plugin. This enables you to use auto-completion feature of your favorite R IDE (like RStudio) and not have to remember or look up supported tile providers; just type

`providers$` and choose from one of the options. You can also use `names(providers)` to view all of the options. Notice how the names of the tiles appear.

The XYZ protocol exposes maps as images for portions of the Earth we will call tiles. The XYZ name stands from the "coordinates" used to locate a given tile. This of the entire planet split up into squares, each of them available with a unique combination of X and Y numbers. Now add a third one (Z) for the zoom level: lower values use less tiles to cover the world, while higher resolution levels (higher Z) will cover progressively smaller areas, but with more detail. Most XYZ APIs expose tiles directly over HTTP, which means we can access them from the browser.

```
library(leaflet)
# To see the first 5 provider tiles
names(providers[1:5])
```

```
[1] "OpenStreetMap"         "OpenStreetMap.Mapnik" "OpenStreetMap.DE"
[4] "OpenStreetMap.CH"      "OpenStreetMap.France"
```

If you want to see the tiles of only one provide you can use the `str_detect` function.

```
library(tidyverse)
# To see all the Open Street Map tiles
names(providers)[str_detect(names(providers), "OpenStreetMap")]
```

```
[1] "OpenStreetMap"         "OpenStreetMap.Mapnik" "OpenStreetMap.DE"
[4] "OpenStreetMap.CH"      "OpenStreetMap.France" "OpenStreetMap.HOT"
[7] "OpenStreetMap.BZH"
```

To add a basemap, we just define the tile with addProviderTiles()

```
leaflet() %>%
  # addTiles()
  addProviderTiles("Stamen.TonerLite")
```

Zooming to a default map view

```
leaflet() %>%
  # addTiles()
  addProviderTiles("Stamen.TonerLite") %>%
  # define set view with coordinates
  setView(lng = -2.967212, lat = 53.406045, zoom = 13)
```

Adding markers and popups (tooltips) and changing

```
popup = c("Tom", "Kendall", "Sean", "Zachary", "Karla")
leaflet() %>%
  addProviderTiles("NASAGIBS.ViirsEarthAtNight2012") %>%
  addMarkers(lng = c(-3.2031323, -0.2416811, -3.4924087, -4.3725404, -2.6607571),
             lat = c(53.4118332, 51.5285582, 55.940874, 55.8553807, 51.4684681),
             popup = popup)
```

Plotting multiple points and storing the map as an R object

```
# Built a dataframe with tibble
hometown <- tibble(
student = c("Tom", "Kendall", "Sean", "Zachary", "Karla", "Lois"),
lon = c(-3.2031323, -0.2416811, -3.4924087, -4.3725404, -2.6607571, -1.6395383),
lat = c(53.4118332, 51.5285582, 55.940874, 55.8553807, 51.4684681, 53.3956347))
leaflet() %>%
addProviderTiles("Stamen.TonerLite") %>%
  # Add markers according to dataframe
addMarkers(lng = hometown$lon, lat = hometown$lat)
```

For some extra help with `leaflet` in R have a look here.

4.2.2 Mapbox Static Tiles API

The Mapbox Static Tiles API serves raster tiles generated from Mapbox Studio styles. Raster tiles can be used in traditional web mapping libraries like Mapbox.js, Leaflet, OpenLayers, and others to create interactive slippy maps. The Static Tiles API is well-suited for maps with limited interactivity or use on devices that do not support WebGL.

```
# R Interface to 'Mapbox' Web Services
library(mapboxapi)
```

```
Usage of the Mapbox APIs is governed by the Mapbox Terms of Service.
Please visit https://www.mapbox.com/legal/tos/ for more information.
```

```
# my_token <- "PLACE YOUR MAPBOX TOKEN HERE and UNCOMMENT"
# mb_access_token(my_token, overwrite = TRUE, install = TRUE)
# readRenviron("~/.Renviron")

leaflet() %>%
```

```
addMapboxTiles(style_id = "light-v9", username = "mapbox" ) %>%
setView( lng =-2.973286, lat = 53.406872, zoom = 13 )
```

You could also call on a basemap you made yourself as shown in the Data Architecture of the course.

EXERCISE

- Explore different basemaps with addProviderTiles() in the `leaflet` library or with `mapboxapi`
- Set a fixed boundary with the function `fitBounds()` and `setMaxBounds()`. You can explore bounding boxes (coordinates) here - Think about data you would could plot on it and why.
- When selecting a Basemap ask yourself some questions
  - Why are you making this map?
  - Is it just for your use or within a bigger project?
  - What type of data will you be plotting?
- In pairs , present your choice of basemap and webmap idea to your partner.

Some Extras - `Leaflet.extras2` has some nice additions to the `leaflet` library. For example, you can integrate easy slide views between two maps. - `Mapview` is a great library that generates interactive maps with **very little code**. You can find tutorials here and here

4.2.3 Directions API

We will explore an API that allows us to tap into the output of computations that take place in the cloud, rather than a direct database. In particular, we will play with the Mapbox Directions API. You will need your mapbox token again

Mapboxapi supports the use of Mapbox's Directions, Isochrone, Matrix, and more, and are designed to be incorporated into R analysis workflows using sf, Shiny, and other packages.

The `mb_directions()` function computes a route between an origin and destination, or along multiple points in an `sf object`. Output options include the route or the route split by route legs as an `sf linestring`, or the full routing output as an `R list` for additional applications.

The general structure of the call is as follows:

```
my_route <- mb_directions( origin = "140 Chatham St, Liverpool L7 7BA", destination = "4 S

leaflet(my_route) %>%
  addMapboxTiles( style_id = "light-v9", username = "mapbox" ) %>%
  addPolylines()
```

It can even give us directions - in multiple languages

```
my_route$instruction
```

```
 [1] "Head north on Chatham Street"
 [2] "Turn right onto Myrtle Street"
 [3] "Continue straight to stay on Myrtle Street"
 [4] "Turn left onto Melville Place"
 [5] "Turn right onto Oxford Street"
 [6] "Continue onto Grinfield Street"
 [7] "Continue onto Chatham Place"
 [8] "Continue onto Harbord Street"
 [9] "Turn left onto Chatsworth Drive"
[10] "Turn left onto Wavertree Road (B5178)"
[11] "Turn right onto Dorothy Drive"
[12] "Turn right onto Royston Street"
[13] "Turn left onto Durning Road (B5173)"
[14] "Turn right onto Edge Lane (A5047)"
[15] "Turn left onto Gresham Street"
[16] "Continue straight to stay on Gresham Street"
[17] "Continue straight onto Gresham Street"
[18] "Turn right onto Edge Grove"
[19] "Turn left onto Stanley Street"
[20] "You have arrived at your destination, on the left"
```

Exercise

- Explore the documentation and play around with some of the `mb_directions()`. Which other `profiles` can you pick? try out some languages for example by adding `language = "fr"`
- Explore the documentation for the isochrone on mapboxapi and try to obtain results - `mb_isochrone()`. For example, retrieve the area that can be reached within 15 minutes of the Roxby Building. Isochrones are areas reachable within a given travel time, around a given location.
- Play around with travel-time matrices with the mb_matrix() function

Note that there are other routing APIs available such as `library(osrm)`.

4.2.4 Geographic Data through APIs and the web

We've share spatial data through APIs. Let's now have a look at how APIs can help us generate and create spatial data.

In the Web Architecture section of the module, you already had a look at API requests. We used both:

- **Writing our own API request.** The `GET` function from `httr` package.
- **Plug-n-play packages.** `get` functions available through user-written R Packages

There are many APIs where we can `GET` data these days. A few examples are:

- US CENSUS API
- Overpass API - Open Street Map
- Transport for London API
- The London DataStore API
- Thames Water API
- London Air API
- Crime data

Another good source of data is the CDRC

Let's go back to the Bike Points example we starting looking at in the Web's architecture session.

```
library(httr)
library(jsonlite)

#key <- "YOURKEY HERE"

request <- GET("https://api.tfl.gov.uk/BikePoint/") # Here we request all the bike docking
```

Checking the Status Code

```
# The response status is 200 for a successful request
request$status_code
```

```
[1] 200
```

Extracting the data frame

```
bikepoints <- jsonlite::fromJSON(content(request, "text")) # extract the dataframe
names(bikepoints) # Print the column names
```

```
 [1] "$type"                "id"                   "url"
 [4] "commonName"           "placeType"            "additionalProperties"
 [7] "children"             "childrenUrls"         "lat"
[10] "lon"
```

```
bikepoints$`Station ID` = as.numeric(substr(bikepoints$id, nchar("BikePoints_")+1, nchar(b
```

Creating an sf object from longitude latitude in the bike dataframe.

```
library(dplyr)
library(sf)

# create a sf object and set the CRS
stations_df <- bikepoints %>%
  sf::st_as_sf(coords = c(10,9))  %>%  # create pts from coordinates
  st_set_crs(4326) %>%  # set the original CRS
  relocate(`Station ID`) # set ID as the first column of the dataframe
```

Now let's add some data about trips made by hire bikes. We need to use the station IDs for the beginning and end of the trips. Transport for London publishes online all trips made by hire bikes along many other datasets related to bike usage in London. The files are published weekly. They have information on starting and ending stations, exact time of the trips.

We can download the files for August 2018 and do some cleaning to map the most used routes in London. We first need to filter for completed trips and select trips with different origins/destinations.

The next step is to aggregate the trips by pairs of origin and destination stations. The results should be how many trips have originated and ended from a specific pair in August 2018.

```
# Uncomment the following to download the trips taken by hire bikes in August 2018

#download.file("https://cycling.data.tfl.gov.uk/usage-stats/121JourneyDataExtract01Aug2018
#              destfile = "data/London/121JourneyDataExtract01Aug2018-07Aug2018.csv")
#download.file("https://cycling.data.tfl.gov.uk/usage-stats/122JourneyDataExtract08Aug2018
```

```
#              destfile = "data/London/122JourneyDataExtract08Aug2018-14Aug2018.csv")
#download.file("https://cycling.data.tfl.gov.uk/usage-stats/123JourneyDataExtract15Aug2018
#              destfile = "data/London/123JourneyDataExtract15Aug2018-21Aug2018.csv")
#download.file("https://cycling.data.tfl.gov.uk/usage-stats/124JourneyDataExtract22Aug2018
#              destfile = "data/London/124JourneyDataExtract22Aug2018-28Aug2018.csv")

# list the cycle hire extracts from TfL
# https://cycling.data.tfl.gov.uk/
library(data.table)
extracts <- list.files("data/London", pattern=glob2rx("*Journey*Data*Extract*"),
                       recursive = TRUE,
                       full.names = TRUE)

# loop through files
journeys <- do.call("rbind", lapply(extracts, fread))

# aggregate at the station day level
journeys_agg <- journeys %>%
  filter(!`StartStation Id`==`EndStation Id`) %>% # filter trip with same origin and desti
  filter(!is.na(`EndStation Id`)) %>% # filter lost bike
  filter(!is.na(`StartStation Id`)) %>% # filter lost bike
  filter(`StartStation Id` %in% stations_df$`Station ID`) %>% # filter stations that close
  filter(`EndStation Id` %in% stations_df$`Station ID`) %>%  # filter stations that closed
  filter(!Duration <= 0) %>% # filter no trips and lost
  filter(Duration <= 180*60) %>%  # filter trips not well docked
  group_by(`StartStation Id`, `EndStation Id`) %>%
  summarise(journeys = n(),
            mean_duration = mean(Duration)) %>%
  ungroup() %>%
  mutate(share_trips = 100*journeys/sum(journeys))

# quick stats
summary(journeys_agg)
```

```
StartStation Id EndStation Id      journeys       mean_duration
Min.   :  1.0   Min.   :  1.0   Min.   :  1.000   Min.   :   60.0
1st Qu.:166.0   1st Qu.:172.0   1st Qu.:  1.000   1st Qu.:  772.5
Median :341.0   Median :349.0   Median :  2.000   Median : 1110.0
Mean   :373.7   Mean   :380.2   Mean   :  4.803   Mean   : 1290.7
3rd Qu.:580.0   3rd Qu.:589.0   3rd Qu.:  5.000   3rd Qu.: 1520.0
Max.   :833.0   Max.   :833.0   Max.   :641.000   Max.   :10800.0
 share_trips
```

```
Min.   :0.0001183
1st Qu.:0.0001183
Median :0.0002365
Mean   :0.0005680
3rd Qu.:0.0005913
Max.   :0.0758079
```

Most origin/destination pairs have 4.8028878 trips during the period. The average duration is 21.5119169 min.

We can then filter our journeys to the top 2 percentiles of the trips. Most pairs do not have any trips (none goes from the furthest station in Hackney down to Oval station). Plotting all lines would be messy.

```
library(stplanr)
# filter out top 2%
od_top2 = journeys_agg %>%
  arrange((journeys)) %>%
  top_frac(0.02, wt = journeys)

# Creating centroids representing desire line start and end points.
desire_lines = od2line(od_top2, stations_df) # here using package stplanr
```

We plot the top 0.2% of pairs by the number of trips (you can reduce the percentage if your computer is too slow). We can see that most trips originate from the centre. Let's try to make it nicer and more interactive:

```
library(classInt)
library(tmap)
# find the breaks
brks <- classIntervals(desire_lines$journeys, 5, style = "jenks")

# plot
tmap_mode("view")
tm_basemap() + # add a London basemap
tm_shape(desire_lines) + # add the OD lines
  tm_lines(id = "journeys", # set the pop up id to the number of journeys
           palette = "plasma", # purple to yellow palette
           breaks = brks$brks, # jenks breaks defined earlier
           lwd = "share_trips", # share trips colour
           scale = 9,
           title.lwd = "Share trips (%)", # set thickness of lines
```

```
          alpha = 0.3, # transparency
          col= "journeys", # set colour fill to number of journeys
          title = "Number of trips"
  ) +
  tm_shape(stations_df) + # add the stations for context
  tm_symbols(id = "commonName", col = "red", alpha = 0, scale = .5) + # names of stations
  tm_scale_bar() +
  tm_layout(
    legend.bg.alpha = 0.5,
    legend.bg.color = "white") # legend format
```

Exercise

- Look at the documentation for one of the APIs mentioned for data extraction (or another API you are aware of) or the CDRC
- Think of 2 ideas of web maps you could construct from your chosen data. Think about the basemap and data you would plot on top.

4.2.5 Geocoding API

Below is a short exploration of a the geocoding API `library(tidygeocoder)`. In your own time try to use it to automatically embed coordinates between addresses.

```
library(tidygeocoder)

# Create a dataframe with addresses
some_addresses <- tibble::tribble(
~name,                  ~addr,
"South Campus Teaching Hub",          "140 Chatham St, Liverpool L7 7BA",
"Sefton Park", "Sefton Park, Liverpool L17 1AP",
"Stanley Street", "4 Stanley St, Liverpool L1 6AA"
)

# Geocode the addresses
lat_longs <- some_addresses %>%
  geocode(addr, method = 'osm', lat = latitude , long = longitude)
```

```
# You could also be reading addressed from a file
liverpool_addresses <- read_sf("data/example_addresses_liverpool.csv")

lat_longs <- liverpool_addresses %>%
```

```
  geocode(addr, method = 'osm', lat = latitude , long = longitude)
```

Reverse Geocoding

You can also reverse geo-code the data, open the output and see what the result is

```
reverse <- lat_longs %>%
  reverse_geocode(lat = latitude, long = longitude, method = 'osm',
                  address = address_found, full_results = TRUE) %>%
  select(-addr, -licence)
```

Other packages also do geocoding such as `library(ggmap)` have a look here

4.3 References

Arribas-Bel, D. (2014) "Accidental, Open and Everywhere: Emerging Data Sources for the Understanding of Cities". Applied Geography, 49: 45-53.

Goodchild, M. F. (2007). Citizens as sensors: the world of volunteered geography. GeoJournal, 69(4), 211-221.

Lazer, D., & Radford, J. (2017). Data ex machina: introduction to big data. Annual Review of Sociology, 43, 19-39.

Leaflet for R

Titorchul, O. (2020), Breaking Down Geocoding

5 Map design

Dani Arribas-Bel

Lecture: Map design

Lab: Designing maps with Mapbox Studio

5.1 Lecture

Slides can be downloaded here

5.2 Lab: Designing maps with Mapbox Studio

In this lab, we will put to use some of the ideas and concepts we have learnt about map design. We will do so using Mapbox Studio, a tool that will allow us to control almost any thinkable aspect of a map.

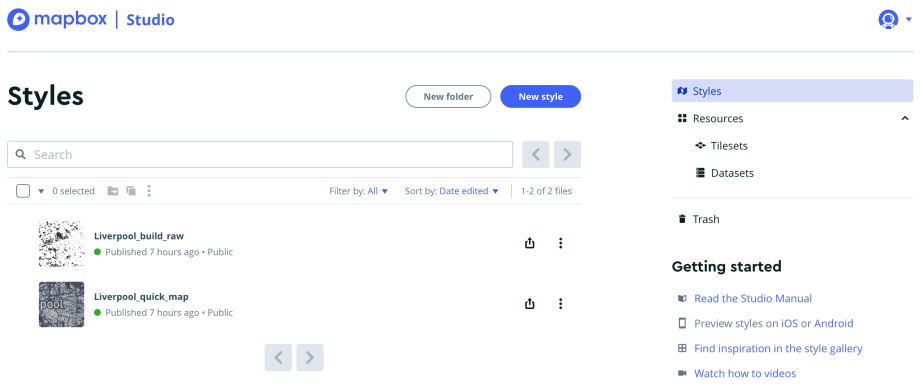
As we go through the practicalities, remember the concepts that inspire map design, which we have outlined in the lecture slides. The challenge here is not in learning the software, but in being able to translate abstract notions of design into an applied context.

To complete this lab, you will require the following:

- The internet
- An active Mapbox account Mapbox Studio

Mapbox Studio

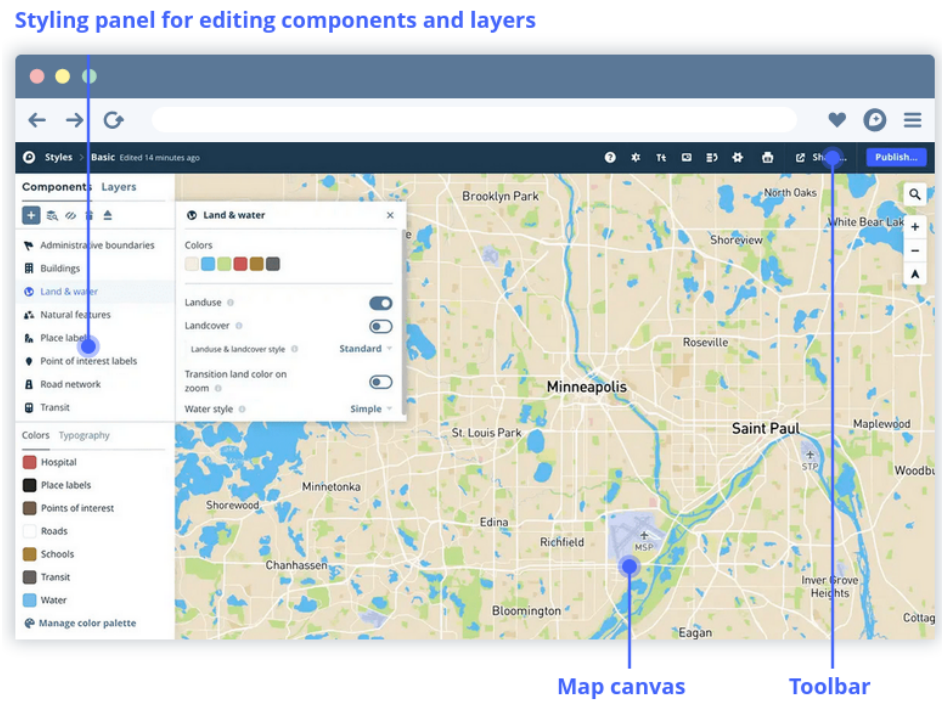
Let's get starting by logging into our Mapbox account and opening up Mapbox Studio



You should see something like this:

To explore what is possible, we will create a new style. Click on the "New Style" button and pick the "Monochrome" option. Select the color you prefer. This will load the Studio editor, where we will spend most of this session.

The Studio is structured around three main panels: styling on the left, the map canvas, and the toolbar at the top.



Source: Mapbox

Most of the time, we will select views, layers, and data from the styling panel, and our actions will drive changes on the map canvas. In this tutorial, we will examine how to modify and style the different elements of design we have seen in class:

- Color
- Texture
- Labelling and typography
- Iconography

We will finish our tour discussing the Elements feature and learning about how to add our own data to the maps we style in Studio.

Exercise

For this exercise, we will be using the "CLIWOC Slim and Routes" data product: https://figshare.com/articles/CLIWOC_Slim_and_Routes/11941224

Team in groups of two to four and pick one of the following options:

1. Global map of trade routes
2. Map of ship activity around South Africa
3. Map showing how important the island of Saint Helena was in this period
4. Map of shipping activity in the English Channel
5. Map of expeditions into the Arctic
6. Map of activity around the Caribbean

Style a map according to its goal.

5.2.1 Bring your mapped style back into R

Go back to the end of the Lab 3 and use the `mapdeck()` function to bring your styled map back into R. For more on mapdeck check this page.

5.2.2 Presentation

Once you are happy with your final style, publish it and drop the link on the module's Team. Designate one member of the group to present it. The presentation should cover the following:

- What is the map about?
- What design elements did you tweak? How? Why?
- What choices did you make following design principles? What other alternatives did you consider? Why did you opt for those choices?

5.3 References

- Katie Jolly's Map Design Guide
- Mapbox's Guide to Map Design
- The Mapbox Studio manual
- The seven deadly sins of visualisation, by James Cheshire
- Wes Anderson color palettes
- The Ship Map, Kiln is a stunning example

6 Interactivity

Elisabetta Pietrostefani

Lecture: Interactivity

Lab: Designing for interactivity with Kepler.gl

6.1 Lecture

Slides can be downloaded here

6.2 Lab

In this lab, we are going to get our hands dirty and play with different elements that allow us to make a map interactive. We will do so using Kepler/gl, which makes possible (and even fun!) to work interactivity into a map in different ways.

As we go through the hands-on aspect of this block, please keep the lecture slides handy and revisit them as much as you need to. As always, the real challenge is not to learn how to use a piece of software, but how to apply conceptual notions in a practical context.

To complete this lab, you will require the following:

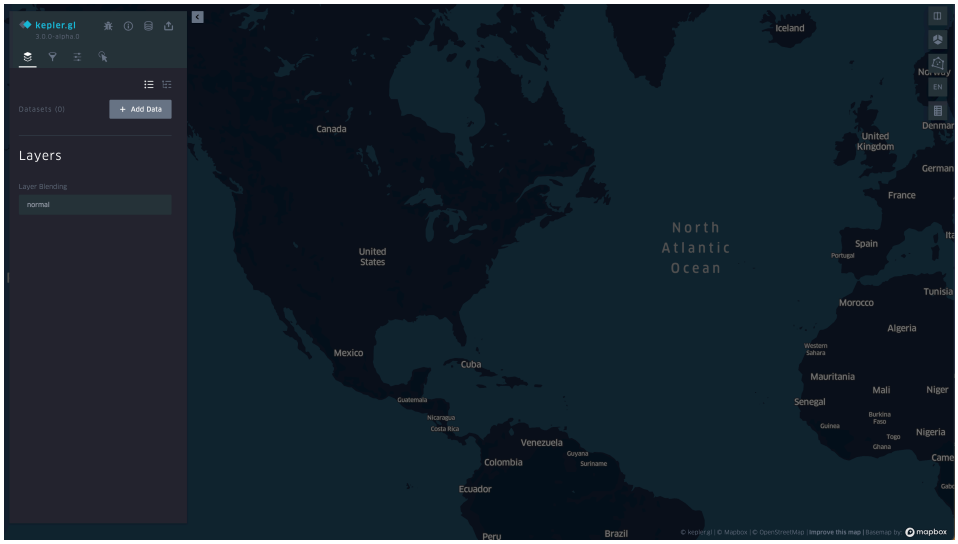
- The internet
- Kepler.gl

6.2.1 Getting to know Kepler.gl

We will use Kepler to quickly be able to make web maps and explore how you can build interactivity in web maps.

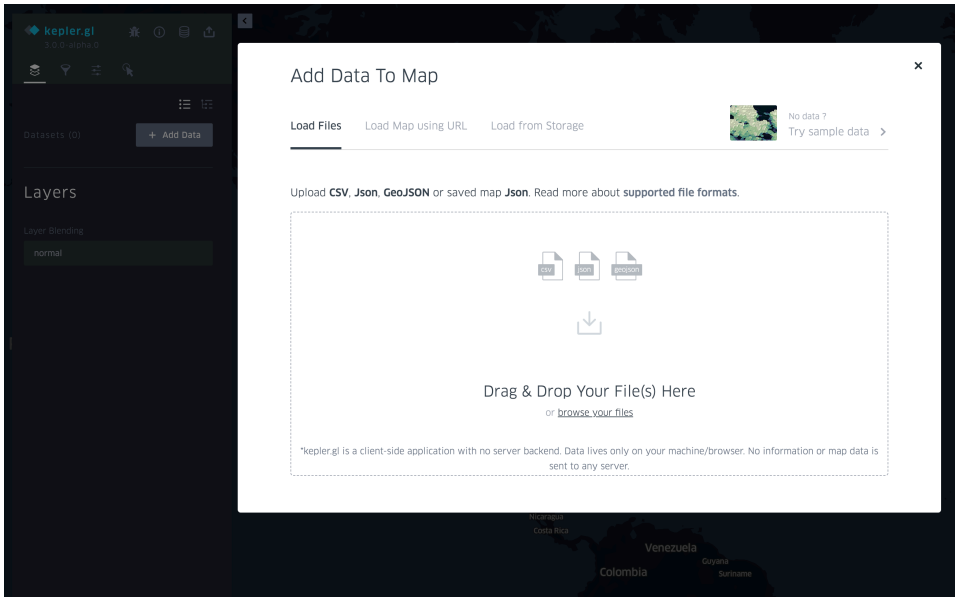
Let us start by open up Kepler.gl

You should see a dashboard that looks more or less like:

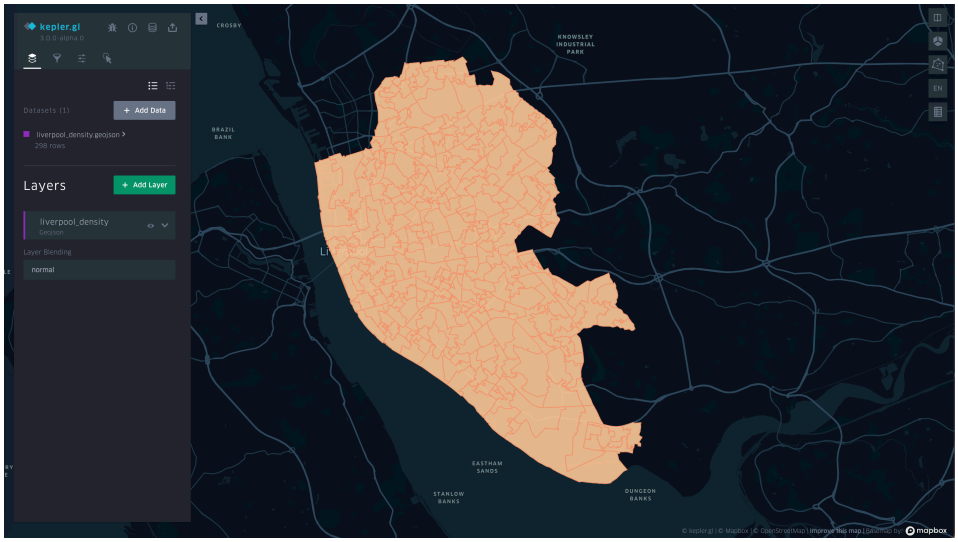


To start understaning the main features, you can create a "New Map" (either from the dasboard or the "Maps" section, you will be able to find that button), and add the imd2019 dataset we used for Lab 6 (remember, you will find it on the "Shared with you" tab). This will take you to a new page that looks roughly like:

We will use some density data you can download here

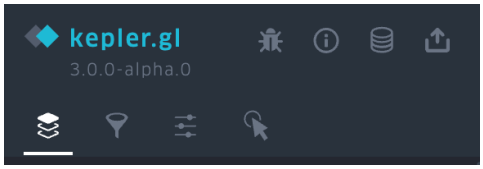


Once the dataset has been uploaded, it shows under Layers on the left-hand side panel.



Explore the Kepler.gl interface.

Let's walk through the basics of Kepler.gl as a (web) GIS:



- Basemap: pick your background
- Main layer: focus on style for now
  - STYLE: fill colour, outline, radius
  - POP-UP: Tooltips
  - LEGEND

There is lots of documentation to help you here.

6.2.2 Interactivity

Now let's remember the building blocks of interactivity we have learnt in the lecture. We will demonstrate in bold those that we will work through in the lab:

- Filtering
  - **Pan**

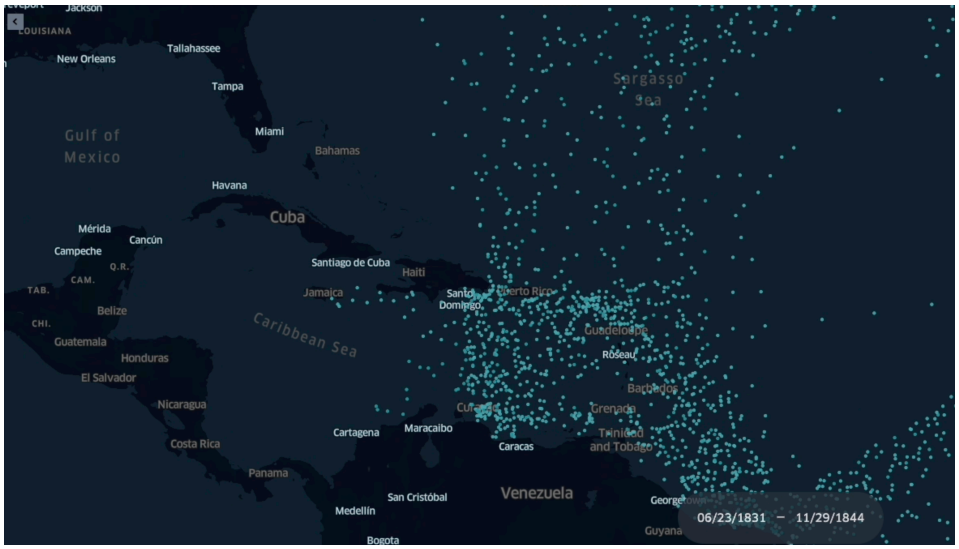
  - **Zoom**
  - **Subset**

- Perspective
- Volume
- **Tooltips**
- Split
- **Animate**

To demonstrate animations, we will use another dataset we have enountered in the past, the CLIWOC ship logs:

https://figshare.com/articles/CLIWOC_Slim_and_Routes/11941224

We will work with individual logs (cliwoc_slim) to create an animation of the logs, for example just showing expeditions between certain date, as in this example of expeditions into the Caribbean.

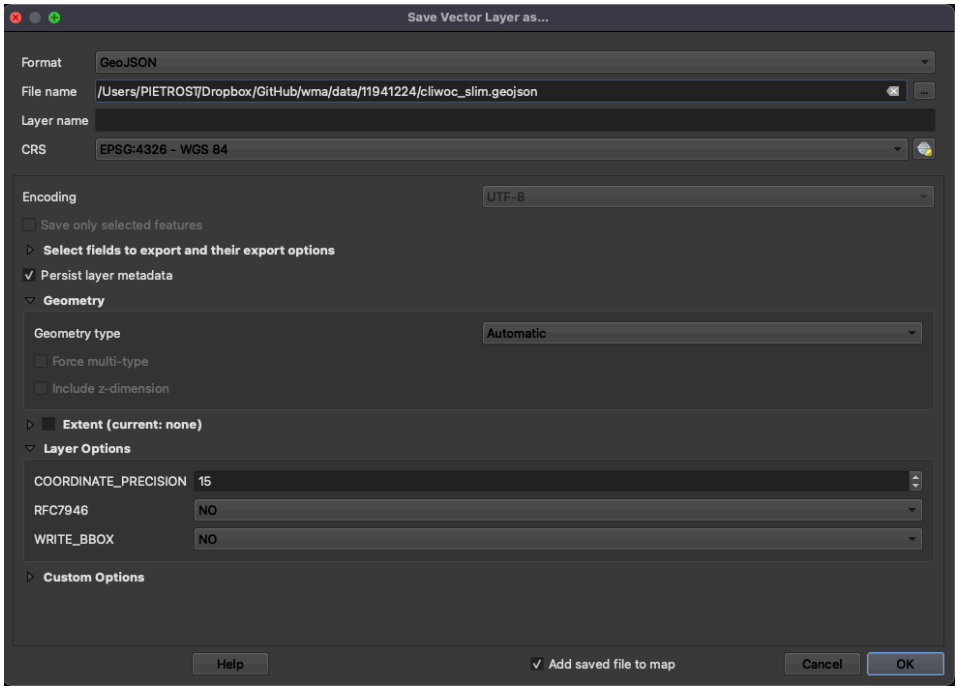


Date Prep

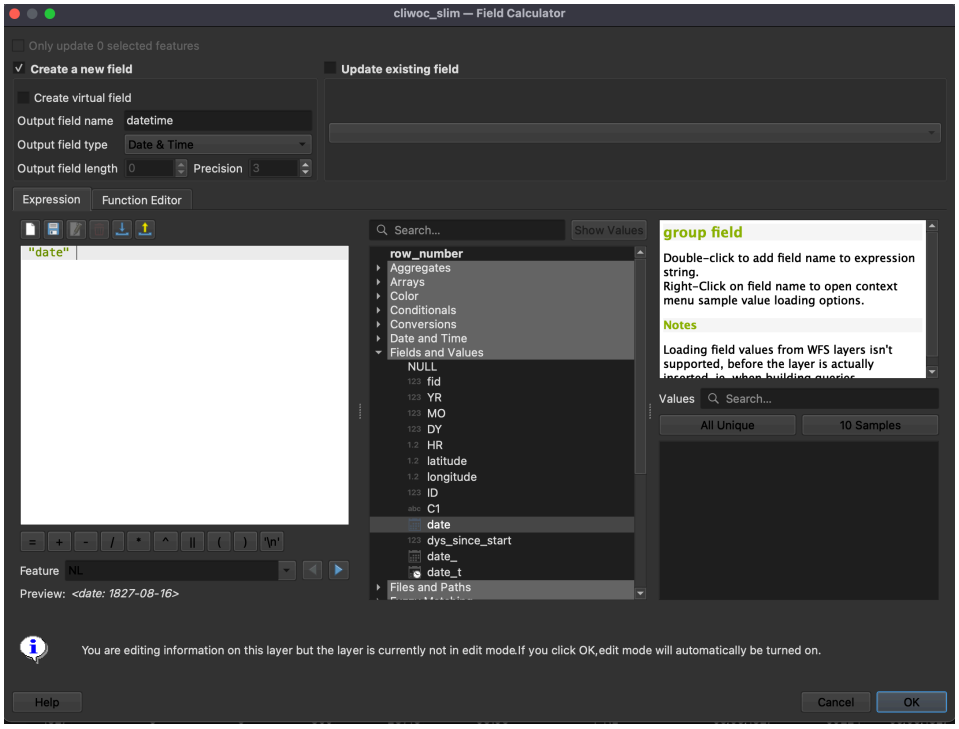
First we need the data in a geojson format and to make sure that the time variable is in the correct date-time format. For this we will need to rely on QGIS or R.

In QGIS there are two steps:

1. Import the cliwoc_slim.geopackage as save as geojson

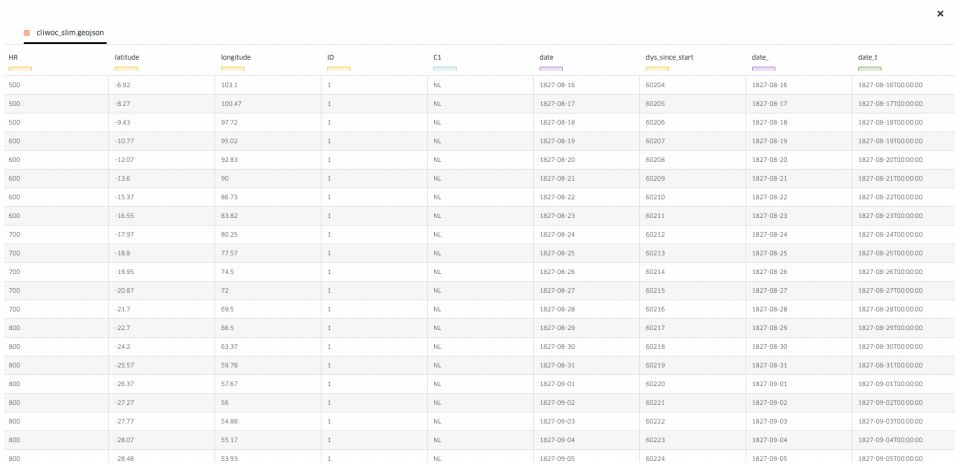


2. Change the date variable to a date and time variable. If you don't do this, kepler.gl won't recognize it as a date.

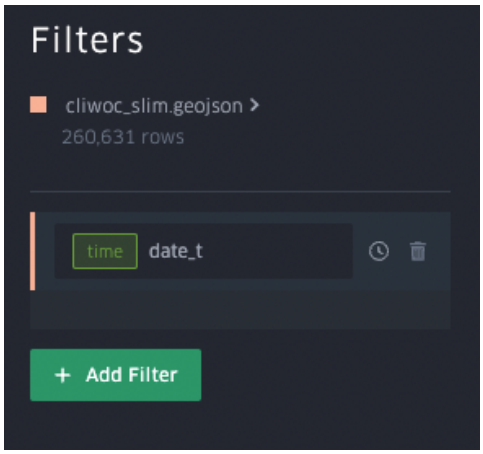


Back to Kepler.gl

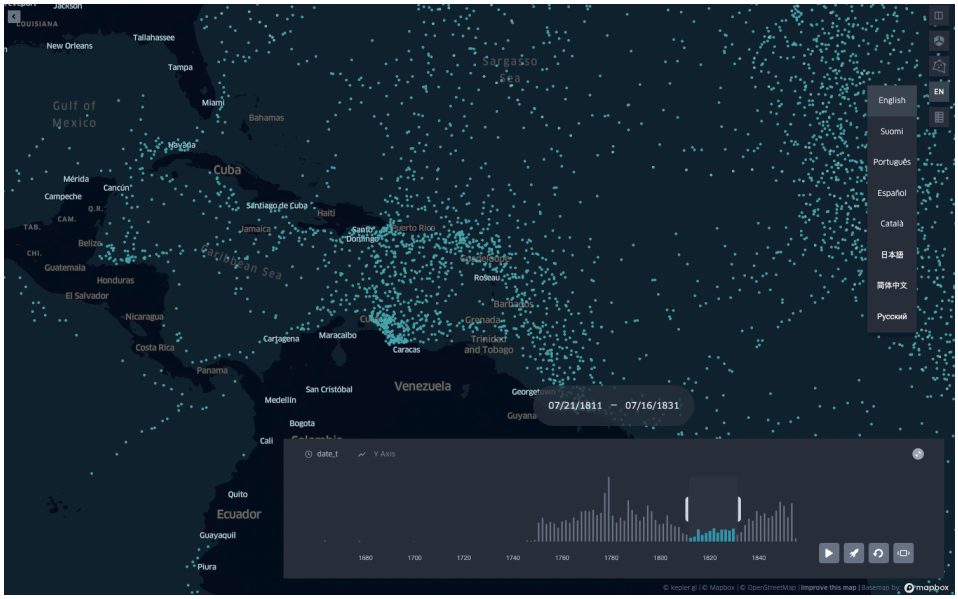
- Once this is done you can import that data to Kepler.gl.
- Inspect the data, focusing on the columns (attributes).



- Create a filter by date (or additional filters).



Notice the time slider showing on the lower right corner.



Exercise

Now we know the mechanics of interactivity in Kepler.gl, let's show off! Pick whichever you want first, and have a go at the following maps:

1. An animation of global trade over time A map that lets you pick a given country and display its main routes
2. A map that lets you identify the vesel ID (id), date, and country of ships around Cape Town
3. A choropleth where you can select routes by their length in days
4. An animation of each route in the region around Jakarta
5. A map that allows you to select a single route, zoom into its origin, and then pan throughout the route

Once completed, select the one you like best, and post it on Teams.

Presentation

- You will then have 30 seconds to present your favorite map and hit the following points:
- What the map shows What interactivity element(s) you have used One thing you think is really effective about it
- Remember, 30 seconds. Short and sweet. Make them count!

6.3 References

- Tamara Munzner's "Visualization Analysis & Design". This lecture draws mostly on Chapter 1 (What's Vis, and Why Do It?).
- Andy Kirk's "Data Visualisation: a Handbook for Data Driven Design".
- Mapbox's Guide to Map Design.
- Geo Temporal data visualisation

7 Statistical visualisation

Dani Arribas-Bel

Lecture: Statistical visualisation

Lab: Choropleths in Kepler or leaflet R

7.1 Lecture

Slides can be downloaded here

7.2 Lab:

7.3 References

8 Dashboards

Dani Arribas-Bel

Lecture: Dashboards: bringing analysis to the web

Lab: Building Dashboards

9 Technology Gallery

Dani Arribas-Bel

Lecture: Technology gallery

Lab: Assignment II clinic.