Breaking Down Geocoding in R: A Complete Guide

How to use APIs to find a place you are interested in and visualize it on a map

Oleksandr Titorchuk

25.04.2020

Contents

0.1 Introduction

If you ever wondered how to build maps similar to the ones you constantly see in your apps, it's probably a good place to start. In this tutorial we will cover how to find a place based on its description or coordinates and how to build a simple map based on that information.

Please note that this article assumes some prior knowledge of R language: data structures, operators, conditional statements, functions etc. All the code from this tutorial can be found on GitHub.

So, let's get started!

0.2 What Is Geocoding?

Geocoding is a process of converting an address or a name of a place into its coordinates. ***Reverse geocoding*** performs just an opposite task: returns an address or a description of a place based on its coordinates.

That's all, as simple as that. So, by using geocoding you must be able to say that *the Eiffel Tower* in Paris, France can be found at (48.858568, 2.294513) latitude, longitude coordinates. And that by entering (41.403770, 2.174379) on your map app, you will end up at *the Sagrada Familia* Roman Catholic church in Barcelona, Spain. You can verify it yourself - just type in this information on Google Maps.

0.3 What Geocoding Tools Are Available?

When it comes to free geocoding tools available online, one of the options is specialized websites. For example, mentioned above Google Maps. After a bit of search, you can find others.

All of these are perfectly fine instruments if you need to find only a few addresses. But imagine there are hundreds of them? And what about thousands? This task rapidly becomes quite a headache.

For bulk requests API is a much more suitable option. And probably the most obvious choice here is Google Maps API. To be able to use Google services, you need to create the account on the Google Cloud Platform and get your API key. Google provides a detailed instruction on how to do it on their website.

Another option is to use a public API from the OpenStreetMap called Nominatim. OpenStreetMap is a collaborative project whose aim is to create a free maps service for the public. As its website says: *"OpenStreetMap is built by a community of mappers that contribute and maintain data about roads, trails, cafes, railway stations, and much more, all over the world."* And Nominatim is basically a tool to power the search on the OpenStreetMap website. Unlike Google Maps, Nominatim does not require you to register any account and get an API key. But if you want to use its services in an application, you need to provide an email address, so your app's activity can be tracked and restrained if needed - OSM servers' capacity has its limits.

0.4 Legal Considerations

You might be wondering why I am telling you about the Nominatim API in the first place if Google offers similar capabilities. And your first guess will probably be the cost - unlike OpenStreetMap Foundation, Google is a private company, which takes charges for its services. And it is true, but only partially.

Firstly, if you register on the Google Cloud Platform now, you are getting a 12-month free trial with $300 credit on your account to learn about its features. Secondly, even after that Google offers a limited access to some of its most commonly used services for free as part of the Always Free package. And if your sole purpose is learning, the limits available under that package are more than enough. To learn more about Google Maps API Pricing, visit Google's help page.

So, what's the issue then you would ask me? It is Google Maps Platform Terms of Service, which among other things state that:

3.2.4 Restrictions Against Misusing the Services.
(a) No Scraping. *Customer will not extract, export, or otherwise scrape Google Maps Content for use outside the Services.*
(c) No Creating Content From Google Maps Content.
(e) No Use With Non-Google Maps.

I am not a legal guy and don't know how Google treats the use of its services for non-commercial purposes. But I haven't seen on these Terms of Service any clause stating that above restrictions apply only to commercial use. So, please be mindful of these limitations before you decide to use Google Maps API in your app.

Unlike Google Maps, OpenStreetMap data is licensed under the Open Data Commons Open Database License (ODbL). Below is, as authors themselves put it, a human-readable summary of ODbL 1.0:

You are free:
To Share: *To copy, distribute and use the database.*
To Create: *To produce works from the database.*
To Adapt: *To modify, transform and build upon the database.*

As long as you:
Attribute: *Give reference to the original database.*
Share-Alike: *Distribute a database adapted from the original one under the same license.*
Keep open: *Give access to the adapted database to the public.*

A full-length license, in case you want to have a look, is available on the Open Data Commons website.

Having said all of that, let's now move on to the coding!

0.5 Installing Packages

Let's firstly install and load all the packages which will be used in this tutorial, so not to worry about it later. The purpose of each package will be described in the corresponding part of the article. Please also note that we are using software version R 3.6.2 for Windows.

```r
# install packages

install.packages("ggmap")
install.packages("tmaptools")
install.packages("RCurl")
install.packages("jsonlite")
install.packages("tidyverse")
install.packages("leaflet")
```

```r
# load packages

library(ggmap)
library(tmaptools)
library(RCurl)
library(jsonlite)
library(tidyverse)
library(leaflet)
```

0.6 Geocoding With R Packages

The R community created a few packages, which can be used for accessing Google Maps and Nominatim APIs. Let's have a look on them.

0.6.1 Package ggmap

The first package is called ggmap and it allows you to connect to the Google Maps API. Before you can start using this package, you need to provide R with your API key.

```r
# replace "api_key" with your API key

register_google(key = api_key)
```

Let's now use `geocode` function from this package on the sample of twelve London pubs to demonstrate how it works. The function accepts as its `output` argument either:

- *latlon* - latitude and longitude;
- *latlona* - all of the above plus address;
- *more* - all of the above plus place's type and geographical boundaries;
- *all* - all of the above plus some additional information.

Each option corresponds to the type of information generated. Generally, we don't need more information than *more* option provides.

```r
# create a list of London pubs

pubs <- c("The Angel, Bermondsey", "The Churchill Arms, Notting Hill", "The Auld Shillelagh, Stoke Newin

pubs_df <- data.frame(Pubs = pubs, stringsAsFactors = FALSE)

# run the geocode function from ggmap package

pubs_ggmap <- geocode(location = pubs, output = "more", source = "google")
pubs_ggmap <- cbind(pubs_df, pubs_ggmap)

# print the results

pubs_ggmap[, 1:6]
```

```
##                                       Pubs        lon      lat          type
## 1                    The Angel, Bermondsey -0.0590456 51.50064           bar
## 2         The Churchill Arms, Notting Hill -0.1948010 51.50691           bar
## 3     The Auld Shillelagh, Stoke Newington -0.0794564 51.56205           bar
## 4                 The Sekforde, Clerkenwell -0.1044597 51.52437 establishment
## 5                    The Dove, Hammersmith -0.2348502 51.49050           bar
## 6  The Crown and Sugar Loaf, Fleet Street -0.1049939 51.51402           bar
## 7                        The Lamb, Holborn -0.1190473 51.52309           bar
## 8           Prince of Greenwich, Greenwich -0.0099570 51.47619           bar
## 9             Ye Olde Mitre, Hatton Garden -0.1073185 51.51841           bar
## 10                  The Glory, Haggerston -0.0770533 51.53620           bar
## 11                   The Blue Posts, Soho -0.1326110 51.51113           bar
## 12   The Old Bank of England, Fleet Street -0.1114818 51.51390           bar
##    loctype                                                   address
## 1  rooftop                      101 bermondsey wall e, london se16 4nb, uk
## 2  rooftop             119 kensington church st, kensington, london w8 7ln, uk
## 3  rooftop 105 stoke newington church st, stoke newington, london n16 0ud, uk
## 4  rooftop                      34 sekforde st, farringdon, london ec1r 0ha, uk
## 5  rooftop                        19 upper mall, hammersmith, london w6 9ta, uk
## 6  rooftop                                   98 fleet st, london ec4y 1dh, uk
## 7  rooftop                    94 lamb's conduit st, holborn, london wc1n 3lz, uk
## 8  rooftop                        72 royal hill, greenwich, london se10 8rt, uk
## 9  rooftop                       1 ely ct, ely pl, holborn, london ec1n 6sj, uk
## 10 rooftop                      281 kingsland rd, haggerston, london e2 8as, uk
## 11 rooftop                          28 rupert st, west end, london w1d 6dj, uk
## 12 rooftop                         194 fleet st, holborn, london ec4a 2lt, uk
```

Let's have a look on our results. As you see we have a pub's name, its coordinates, type of the place, precision of the result (*rooftop* means that Google was able to find the place down to a specific building) and its address.

Now, let's use the coordinates we just found to reverse geocode the places they belong to.

The `revgeocode` function allows to do that. It requires two arguments: `location` - a numeric vector of longitude/latitude and `output` - either *address* or *all*. Option *all* for `output` returns much more information than we need, so let's stick to the *address*.

This time we will store our results in a list rather than a data frame. Each element of this list will contain another list with information about the pub's name, coordinates and address.

```r
# extract the coordinates of London pubs

pubs_ggmap_crd <- list()

for (i in 1:dim(pubs_ggmap)[1]) {

    lon <- pubs_ggmap$lon[i]
    lat <- pubs_ggmap$lat[i]
    pubs_ggmap_crd[[i]] <- c(lon, lat)

}

# reverse geocode the coordinates and save them to the list

pubs_ggmap_address <- list()

for (i in 1:length(pubs_ggmap_crd)) {

    pub <- pubs[i]
    crd <- pubs_ggmap_crd[[i]]
    address <- revgeocode(location = crd, output = "address")
    pubs_ggmap_address[[i]] <- list(pub, crd, address)

}

# print the details of the first pub

pubs_ggmap_address[[1]]
```

```
## [[1]]
## [1] "The Angel, Bermondsey"
##
## [[2]]
## [1] -0.0590456 51.5006419
##
## [[3]]
## [1] "101 Bermondsey Wall E, London SE16 4NB, UK"
```

That's all for the ggmap. Let's now move on to the next package.

0.6.2 Package tmaptools

The tmaptools is a package that offers a set of tools for reading and processing of a spatial data. It facilitates the capabilities of another R package called tmap, which was built for visualizing thematic maps. Many of the tmaptools functions rely on the Nominatim API.

Now let's try to get from tmaptools the same sort of information we extracted by using ggmap. I had to modify a bit some search requests because Nominatim was not able to find the place based on it. And one of the pubs - *The Glory* - couldn't be located despite all my efforts. So, be aware that the quality of data and its completeness might vary among different services providers.

```r
# modifying some search requests

pubs_m <- pubs
pubs_m[pubs_m=="The Crown and Sugar Loaf, Fleet Street"] <- "The Crown and Sugar Loaf"
pubs_m[pubs_m=="Ye Olde Mitre, Hatton Garden"] <- "Ye Olde Mitre"
pubs_m_df <- data.frame(Pubs = pubs_m, stringsAsFactors = FALSE)

# geocoding the London pubs
# "bar" is special phrase added to limit the search

pubs_tmaptools <- geocode_OSM(paste(pubs_m, "bar", sep = " "),
                              details = TRUE, as.data.frame = TRUE)

# extracting from the result only coordinates and address

pubs_tmaptools <- pubs_tmaptools[, c("lat", "lon", "display_name")]
pubs_tmaptools <- cbind(Pubs = pubs_m_df[-10, ], pubs_tmaptools)

# print the results

pubs_tmaptools
```

```
##                                      Pubs      lat         lon
## 1                   The Angel, Bermondsey 51.50064 -0.05906115
## 2        The Churchill Arms, Notting Hill 51.50687 -0.19482219
## 3    The Auld Shillelagh, Stoke Newington 51.56202 -0.07945320
## 4               The Sekforde, Clerkenwell 51.52429 -0.10442487
## 5                   The Dove, Hammersmith 51.49051 -0.23485742
## 6                The Crown and Sugar Loaf 51.51403 -0.10500050
## 7                       The Lamb, Holborn 51.52304 -0.11903103
## 8          Prince of Greenwich, Greenwich 51.47623 -0.00984180
## 9                           Ye Olde Mitre 51.51843 -0.10738277
## 10                   The Blue Posts, Soho 51.51290 -0.13943685
## 11 The Old Bank of England, Fleet Street 51.51391 -0.11146837
##
## 1                       The Angel, 101, Bermondsey Wall East, Bermondsey Spa, Bermondsey, London Borou
## 2                             The Churchill Arms, Campden Street, Notting Hill, Royal Borough of Kensi
## 3  The Auld Shillelagh, 105, Stoke Newington Church Street, South Hornsey, Stoke Newington, London Bo
## 4                                    The Sekforde, Woodbridge Street, Angel, Clerkenwell, London Borou
## 5                                       The Dove, 19, Upper Mall, Brook Green, London Borough of Hamme
## 6                                                                Crown & Sugar Loaf, 26, Bride Lane
## 7                                  The Lamb, 94, Lamb's Conduit Street, Holborn, Bloomsbury, London Bo
## 8                                                The Prince Of Greenwich, 72, Royal Hill, Royal Borou
## 9                                                        Ye Olde Mitre, Ely Place, Holborn, London Bo
## 10                                                           The Blue Posts, Kingly Street, Soho, Ci
## 11                                                        The Old Bank of England, 194, Fleet Street
```

Now, it's time for reverse geocoding. In our output we will display the very same information as in reverse geocoding request from ggmap.

```r
# remove The Glory pub from the list ("s" for "short")

pubs_s <- pubs_m[!pubs_m %in% "The Glory, Haggerston"]
```

```r
# extract the coordinates of London pubs

pubs_tmaptools_crd <- list()

for (i in 1:dim(pubs_tmaptools)[1]) {

    lon <- pubs_tmaptools$lon[i]
    lat <- pubs_tmaptools$lat[i]
    pubs_tmaptools_crd[[i]] <- c(lon, lat)

}

# reverse geocode the coordinates and save them to a list

pubs_tmaptools_address <- list()

for (i in 1:length(pubs_tmaptools_crd)) {

    pub <- pubs_s[i]
    crd <- pubs_tmaptools_crd[[i]]
    address <- rev_geocode_OSM(x = crd[1], y = crd[2],as.data.frame = TRUE)[, c("name")]
    pubs_tmaptools_address[[i]] <- list(pub, crd, address)

}

# print the details about the first pub

pubs_tmaptools_address[[1]]
```

```
## [[1]]
## [1] "The Angel, Bermondsey"
##
## [[2]]
## [1] -0.05906115 51.50063675
##
## [[3]]
## [1] "The Angel, 101, Bermondsey Wall East, Bermondsey Spa, Bermondsey, London Borough of Southwark, 
```

So, that's the last piece of code for our discussion of R geocoding packages. And here you can finish reading this article and practice some of the techniques described above by yourself. Unless... Unless you want to find out more! If that's the case, let's continue!

0.7 Geocoding With API

Using packages is a very convenient and fast way to get things done. And probably for most of the tasks you would want to do, the functionality these packages provide is more than enough. However, if you need something extra, or you are interested in other API functions, or you just want to learn how to work with API in general, you need to go to Google/Nominatim help pages and do a bit of reading. Or search online for some videos/tutorials like this offering short summaries. Or even better - do both.

0.7.1 Google Maps API

Having looked at the ggmap package, let's now try to get the place's location, address, and as a bonus, its phone number and a website, using Google Maps API directly. To accomplish this task we need the Geocoding API and the Places API.

0.7.1.1 Geocoding API The Geocoding API is a service that provides the capabilities of geocoding and reverse geocoding of addresses and places. You can access the Geocoding API by sending HTTP request through your web-browser and get back response in JSON or XML format. Although, in our case we will be sending this request from R.

A Geocoding API requests take the following format.

```
# format
https://maps.googleapis.com/maps/api/geocode/outputFormat?parameters

# geocoding example
https://maps.googleapis.com/maps/api/geocode/json?address=The+Churchill+Arms,+Notting+Hill&key=YOUR_API_

# reverse geocoding example
https://maps.googleapis.com/maps/api/geocode/json?latlng=51.5069117,-0.194801&key=YOUR_API_KEY
```

So, the web request you send consists of several parts - the API url followed by `outputFormat` (json or xml) and the list of `parameters`. `outputFormat` is separated from `parameters` by a question mark (?) and `parameters` itself are separated from each other by an ampersand (&).

The required parameters for geocoding request include:

- `address` - a search query in a form of address or place's name;
- `key` - API key.

The required parameters for reverse geocoding request include:

- `latlng` - latitude and longitude of the place you search;
- `key` - API key.

We will not use any optional parameters in our queries.

You can read more about how to construct API requests here and here.

It's worth to mention that if you are building your own application, which needs real-time access to the Google Maps services, you can check Google client-side (JavaScript) or server-side (Java, Python, Go, Node.js) API.

0.7.1.2 Places API In case you do not want to limit yourself to the place's address and coordinates only, you can use the Places API. For example, to find a phone number and a web address of the place we need to use Place Search to get the Place ID and use it later to retrieve this information from Place Details.

While doing API call, make sure to provide the list of `fields` you want to extract. Otherwise, Google will send all of them and charge you accordingly. In our case it doesn't matter as we will not exceed the charge-free limits but **if you plan to use the API for a large volume of requests you might be charged for that**.

For the Place Search/Place Details you also need to provide the `outputFormat` (json or xml) followed by the list of `parameters.`

For the Place Search the required parameters include:

- `input` - a name, an address or a phone number (coordinates will not work);
- `inputtype` - *textquery* or *phonenumber*;
- `key` - API key.

For the Place Details the required parameters are:

- `place_id` - can be found by using the Place Search;
- `key` - API key.

For both the Place Search and Place Details we will be using optional parameter `fields` - a comma-separated list of additional information we want Google to return. You can read more about possible options on corresponding help pages provided earlier. But in our case we only need fields *place_id* from the Place Search and *formatted_phone_number* plus *website* from the Place Details. **Please remember to read information about the billing!**

The format of API calls is given below.

```
# PLACE SEARCH

# format
https://maps.googleapis.com/maps/api/place/findplacefromtext/outputFormat?parameters

# example
https://maps.googleapis.com/maps/api/place/findplacefromtext/json?input=The+Churchill+Arms,+Notting+Hil

# PLACE DETAILS

# format
https://maps.googleapis.com/maps/api/place/details/outputFormat?parameters

# example
https://maps.googleapis.com/maps/api/place/details/json?place_id=ChIJGTDVMfoPdkgROs9QO9Kgmjc&fields=for
```

And again, if you consider building an actual app, it is worth to have a look at Java/Python/Go/Node.js clients for server-side applications or Places SDK for Android, Places SDK for iOS and the Places Library, Maps JavaScript API for the client-side ones.

0.7.2 Geocoding Using Google Maps API

Having said all of that, let's now write the code itself.

Our code consists of eight functions:

- a main function;
- 1 function for checking arguments of the main function;
- 3 functions for generating API calls used in the main function;
- 3 functions to extract data from JSON output.

Our main function takes 3 arguments:

- `search_query` - search request (address or place) ;
- `fields` - information to extract (*coordinates*, *address*, *contacts* or *all*);

- `key` - API key for Google Maps.

The first and the last of them are required, while the second one is optional with default *coordinates*.

These arguments can be of below types:

- `search_query` - string, character vector, list of characters, one-dimensional matrix or data frame with string data;
- `fields` - string, character vector, list of characters;
- `key` - string.

Depending on the value of `fields` the function returns a data frame with:

- *coordinates* - latitude and longitude;
- *address* - full address and city;
- *contacts* - phone number and website;
- *all* - all of the above.

Let's now have a look at each of the function's components in detail.

0.7.2.1 Checking Arguments

The first suplementary function takes the same arguments as our main function and checks whether they are in a proper format. If not it generates a descriptive error message to be passed to the main function. As this chunk of code is not a core part of our function and we can easily deal without it, I will not explain how it works. But the code itself is given below for you to explore.

```
# ///////////////////////////////////////////////
# SUPLEMENTARY FUNCTIONS
# ///////////////////////////////////////////////

# ///////////////////////////////////////////////
# 1. CHECK ARGUMENTS
# ///////////////////////////////////////////////

check_arguments <- function(search_query_arg, fields_arg, key_arg) {

    # create empty error messages
    msg_1 <- msg_2 <- msg_3 <- ""

    # check if "search_query" argument is a string or
    # a list/vector/data frame with string data

    if(is.data.frame(search_query_arg) || is.matrix(search_query_arg)) {
        is_df <- TRUE
    } else {
        is_df <- FALSE
    }

    if(is_df) {
        if(dim(search_query_arg)[2] > 1) {
            error_1 <- TRUE
        } else if(!is.character(search_query_arg[, 1])) {
            error_1 <- TRUE
        } else {
            error_1 <- FALSE
        }
    }

    if(!is_df && is.list(search_query_arg)) {
        match_1 <- c()
        for(i in 1:length(search_query_arg)) {
            match_1[i] <- !is.character(search_query_arg[[i]])
        }
        if(any(match_1)) {
            error_1 <- TRUE
        } else {
            error_1 <- FALSE
        }
    }

    if(!is_df && !is.list(search_query_arg)) {
        if(!is.character(search_query_arg)) {
            error_1 <- TRUE
        } else {
            error_1 <- FALSE
        }
    }

    if(error_1) {
        msg_1 <- "Error: search_query argument (or any of its elements) is not of a string type"
    }

    # check if "fields" argument:
    # consists of a single word only - "all"
    # OR
    # * is a string vector of length <=3
    # * and consists of words 'coordinates', 'address', 'contacts' only

    match_2 <- fields_arg %in% c("coordinates", "address", "contacts")

    if (!(fields_arg %in% "all" && length(unique(fields_arg)) == 1)) {
            if (!all(match_2) || length(unique(fields_arg)) > 3) {
                    msg_2 <- paste0("Error: fields argument must be ",
                      "either a combination of 'coordinates', ",
                      "'address', and 'contacts' or a single ",
                      "word 'all'")
            }
    }

    # check if "key" argument is a string

    if(is.list(key_arg)) {
        if(length(key_arg) > 1) {
            error_3 <- TRUE
        } else if(!is.character(key_arg[[1]])) {
            error_3 <- TRUE
        } else {
            error_3 <- FALSE
        }
    } else if(!is.character(key_arg) || length(key_arg) > 1) {
        error_3 <- TRUE
    } else {
        error_3 <- FALSE
    }

    if(error_3) {
        msg_3 <- "Error: key argument is not of a string type"
    }

    # return error messages (if any)

    errors <- c(msg_1, msg_2, msg_3)

    if(any(errors!="")) {
        errors <- paste(errors[errors!=""], collapse = "\n")
        return(list(TRUE, errors))
    } else {
        return(list(FALSE))
    }

}

# ////////////////////////////////////////////////
```

Let's test whether this function works as we expect.

List of test cases:

1. `search_query`:
    1. **Success**:
        1. a string;
        2. a character vector;
        3. a list of characters;
        4. a one-dimensional matrix with characters;
        5. a one-dimensional data frame with characters;
    2. **Failure**:
        1. a numeric;
        2. a numeric vector;
        3. a list with numerics;
        4. a one-dimensional matrix with numerics;
        5. a two-dimensional matrix with characters;
        6. a one-dimensional data frame with numerics;
        7. a two-dimensional data frame with characters;
2. `fields`:
    1. **Success**:
        1. a string - "all";

        2. a string - "coordinates";
        3. a character vector - c("all");
        4. a character vector - c("coordinates", "address", "contacts");
        5. a character vector with duplicates - c("all", "all", "all");
        6. a character vector with duplicates - c("coordinates", "address", "address", "address");
        7. a list of characters - list("all");
        8. a list of characters - list("coordinates", "address", "contacts");
        9. a list of characters with duplicates - list("all", "all", "all");
        10. a list of characters with duplicates - list("contacts", "address", "address", "address");

    2. **Failure**:

        1. a numeric;
        2. a numerics vector;
        3. a list with numerics;
        4. a character vector with wrong elements - c("alls");
        5. a character vector with wrong elements - c("coordinate", "address");
        6. a list of characters with wrong elements - list("alls");
        7. a list of characters with wrong elements - list("address", "contact");
        8. a character vector with >1 elements and "all" - c("all", "coordinates");
        9. a character vector with >3 unique elements - c("coordinates", "address", "x", "y");
        10. a list of characters with >1 element and "all" - list("all", "coordinates");
        11. a list of characters with >3 unique elements - list("coordinates", "x", "y", "z");

3. `key`:

    1. **Success**:

        1. a string;

    2. **Failure**:

        1. a numeric;
        2. a character vector;
        3. a list of characters.

I do not expect the above list to be exhaustive and I am sure there are some cases which code below doesn't account for. But I think it still covers more than 90% of wrong input scenarios and is a pretty decent job.

```
# /////////////////////////////
# let's create our inputs first
# /////////////////////////////

search_query_1_1 <- pubs[1]
search_query_1_2 <- pubs
search_query_1_3 <- as.list(pubs)
search_query_1_4 <- as.matrix(pubs)
search_query_1_5 <- as.data.frame(pubs, stringsAsFactors = FALSE)

search_query_2_1 <- 1
search_query_2_2 <- c(1,2,3)
search_query_2_3 <- list(1,2,3)
search_query_2_4 <- as.matrix(c(1,2,3))
search_query_2_5 <- cbind(as.matrix(pubs), as.matrix(pubs))
search_query_2_6 <- as.data.frame(c(1,2,3))
search_query_2_7 <- data.frame(one = pubs, two = pubs, stringsAsFactors = FALSE)
```

```r
fields_1_1 <- "all";
fields_1_2 <- "coordinates"
fields_1_3 <- c("all")
fields_1_4 <- c("coordinates", "address", "contacts")
fields_1_5 <- c("all", "all", "all")
fields_1_6 <- c("coordinates", "address", "address", "address")
fields_1_7 <- list("all")
fields_1_8 <- list("coordinates", "address", "contacts")
fields_1_9 <- list("all", "all", "all")
fields_1_10 <- list("contacts", "address", "address", "address")

fields_2_1 <- 1
fields_2_2 <- c(1,2,3)
fields_2_3 <- list(1,2,3)
fields_2_4 <- c("alls")
fields_2_5 <- c("coordinate", "address")
fields_2_6 <- list("alls")
fields_2_7 <- list("address", "contact")
fields_2_8 <- c("all", "coordinates")
fields_2_9 <- c("coordinates", "address", "x", "y")
fields_2_10 <- list("all", "coordinates")
fields_2_11 <- list("coordinates", "x", "y", "z")

key_1_1 <- "xyz"

key_2_1 <- 1
key_2_2 <- c("x", "y", "z")
key_2_3 <- list("x", "y", "z")

# ////////////////////////////////////
# now let's run some tests for SUCCESS
# ////////////////////////////////////

sq_1_1 <- check_arguments(search_query_1_1, "address", "key")
sq_1_2 <- check_arguments(search_query_1_2, "address", "key")
sq_1_3 <- check_arguments(search_query_1_3, "address", "key")
sq_1_4 <- check_arguments(search_query_1_4, "address", "key")
sq_1_5 <- check_arguments(search_query_1_5, "address", "key")

f_1_1 <- check_arguments("search", fields_1_1, "key")
f_1_2 <- check_arguments("search", fields_1_2, "key")
f_1_3 <- check_arguments("search", fields_1_3, "key")
f_1_4 <- check_arguments("search", fields_1_4, "key")
f_1_5 <- check_arguments("search", fields_1_5, "key")
f_1_6 <- check_arguments("search", fields_1_6, "key")
f_1_7 <- check_arguments("search", fields_1_7, "key")
f_1_8 <- check_arguments("search", fields_1_8, "key")
f_1_9 <- check_arguments("search", fields_1_9, "key")
f_1_10 <- check_arguments("search", fields_1_10, "key")

k_1_1 <- check_arguments("search", "address", key_1_1)

# let's check if all the elements in the array below are FALSE
```

```r
success <- c(sq_1_1[[1]], sq_1_2[[1]], sq_1_3[[1]], sq_1_4[[1]], sq_1_5[[1]],
             f_1_1[[1]], f_1_2[[1]], f_1_3[[1]], f_1_4[[1]], f_1_5[[1]],
             f_1_6[[1]], f_1_7[[1]], f_1_8[[1]], f_1_9[[1]], f_1_10[[1]],
             k_1_1[[1]])

if(length(unique(success)) != 1) {
    FALSE
} else if(unique(success) == TRUE) {
    FALSE
} else {
    TRUE
}
```

```
## [1] TRUE
```

```r
# //////////////////////////////////////
# now let's run some tests for FAILURE
# //////////////////////////////////////

sq_2_1 <- check_arguments(search_query_2_1, "address", "key")
sq_2_2 <- check_arguments(search_query_2_2, "address", "key")
sq_2_3 <- check_arguments(search_query_2_3, "address", "key")
sq_2_4 <- check_arguments(search_query_2_4, "address", "key")
sq_2_5 <- check_arguments(search_query_2_5, "address", "key")
sq_2_6 <- check_arguments(search_query_2_6, "address", "key")
sq_2_7 <- check_arguments(search_query_2_7, "address", "key")

f_2_1 <- check_arguments("search", fields_2_1, "key")
f_2_2 <- check_arguments("search", fields_2_2, "key")
f_2_3 <- check_arguments("search", fields_2_3, "key")
f_2_4 <- check_arguments("search", fields_2_4, "key")
f_2_5 <- check_arguments("search", fields_2_5, "key")
f_2_6 <- check_arguments("search", fields_2_6, "key")
f_2_7 <- check_arguments("search", fields_2_7, "key")
f_2_8 <- check_arguments("search", fields_2_8, "key")
f_2_9 <- check_arguments("search", fields_2_9, "key")
f_2_10 <- check_arguments("search", fields_2_10, "key")
f_2_11 <- check_arguments("search", fields_2_11, "key")

k_2_1 <- check_arguments("search", "address", key_2_1)
k_2_2 <- check_arguments("search", "address", key_2_2)
k_2_3 <- check_arguments("search", "address", key_2_3)

# let's check if all the elements in the array below are TRUE

failure <- c(sq_2_1[[1]], sq_2_2[[1]], sq_2_3[[1]], sq_2_4[[1]], sq_2_5[[1]],
             sq_2_6[[1]], sq_2_7[[1]], f_2_1[[1]], f_2_2[[1]], f_2_3[[1]], f_2_4[[1]],
             f_2_5[[1]], f_2_6[[1]], f_2_7[[1]], f_2_8[[1]], f_2_9[[1]], f_2_10[[1]],
             f_2_11[[1]], k_2_1[[1]], k_2_2[[1]], k_2_3[[1]])

all(failure)
```

```
## [1] TRUE
```

```r
# let's see if error messages are generated correctly

failure_sq <- unique(c(sq_2_1[[2]], sq_2_2[[2]], sq_2_3[[2]], sq_2_4[[2]], sq_2_5[[2]],
                sq_2_6[[2]], sq_2_7[[2]]))
failure_f <- unique(c(f_2_1[[2]], f_2_2[[2]], f_2_3[[2]], f_2_4[[2]], f_2_5[[2]],
               f_2_6[[2]], f_2_7[[2]], f_2_8[[2]], f_2_9[[2]], f_2_10[[2]], f_2_11[[2]]))
failure_k <- unique(c(k_2_1[[2]], k_2_2[[2]], k_2_3[[2]]))

messages <- list(failure_sq, failure_f, failure_k)

messages
```

```
## [[1]]
## [1] "Error: search_query argument (or any of its elements) is not of a string type"
##
## [[2]]
## [1] "Error: fields argument must be either a combination of 'coordinates', 'address', and 'contacts'
##
## [[3]]
## [1] "Error: key argument is not of a string type"
```

```r
# and finally let's see how several error messages are displayed at once

cat(check_arguments(c(1,2,3), "contact", 1)[[2]])
```

```
## Error: search_query argument (or any of its elements) is not of a string type
## Error: fields argument must be either a combination of 'coordinates', 'address', and 'contacts' or a
## Error: key argument is not of a string type
```

0.7.2.2 Generating API Call

The API call function is pretty straightforward once you get acquainted with the information I provided earlier.

It works in three steps:

1. Transforms the search query into a list.
2. Percent-encodes the search query.
3. Constructs the API call string.

The first step is needed because it's always easier to deal with a common data structure (a list in our case). For percent-encoding we use `URLencode` function from RCurl package. If you don't know what it is, visit this page with detailed explanation.

```r
# ////////////////////////////////////////////////
# SUPLEMENTARY FUNCTIONS
# ////////////////////////////////////////////////

# ////////////////////////////////////////////////
# 2. GENERATE API CALLS
# ////////////////////////////////////////////////

url_google_geocoding <- function(search_query_url, key_url) {
    # load libraries
    library(RCurl)

    # convert input into a list
    search_query_url <- sapply(search_query_url, as.list)

    # google gecoding api url
    url_geocoding_api <- "https://maps.googleapis.com/maps/api/geocode/"

    # percent-encode search request
    search_query_url <- sapply(search_query_url, URLencode)

    # construct search request for geocode
    url_geocoding_call <- paste0(url_geocoding_api, "json",
                                 "?address=", search_query_url, "&key=", key_url)

    return(url_geocoding_call)

}

# ////////////////////////////////////////////////

url_google_place_search <- function(search_query_url, key_url) {

    # load libraries
    library(RCurl)

    # convert input into a list
    search_query_url <- sapply(search_query_url, as.list)

    # google places api url
    url_places_api <- "https://maps.googleapis.com/maps/api/place/"

    # percent-encode search request
    search_query_url <- sapply(search_query_url, URLencode)

    # construct search request for place id
    url_place_search_call <- paste0(url_places_api, "findplacefromtext/",
                                    "json", "?input=", search_query_url,
                                    "&inputtype=textquery","&fields=place_id",
                                    "&key=", key_url)

    return(url_place_search_call)

}

# ////////////////////////////////////////////////

url_google_place_details <- function(place_id_url, key_url) {

    # load libraries
    library(RCurl)
    # google places api url
    url_places_api <- "https://maps.googleapis.com/maps/api/place/"

    # in case you would want to add "fields" as an argument
    # fields_url <- paste(fields_url, collapse = ",")

    # construct search request for place details
    url_place_details_call <- paste0(url_places_api, "details/",
                                     "json", "?place_id=", place_id_url,
                                     "&fields=formatted_phone_number,website",
                                     "&key=", key_url)

    return(url_place_details_call)

}

# ///////////////////////////////////////////////
```

0.7.2.3 Extracting Data From JSON Google can return data in two formats - JSON and XML. In our examples we use JSON output. This output needs to be converted into R objects, so we can easily manipulate the data it contains. Once it's done, our task comes down to picking up from the formatted list only the elements we need.

Raw JSON output and its formatted version look like this.

```
{
   "results" : [
      {
         "address_components" : [
            {
               "long_name" : "119",
               "short_name" : "119",
               "types" : [ "street_number" ]
            },
            {
               "long_name" : "Kensington Church Street",
               "short_name" : "Kensington Church St",
               "types" : [ "route" ]
            }
         ],
         "formatted_address" : "119 Kensington Church St, Kensington, London W8 7LN, UK",
         "geometry" : {
            "location" : {
               "lat" : 51.5069117,
               "lng" : -0.194801
            },
         },
      }
   ],
   "status" : "OK"
}
```

```
$results
$results[[1]]
$results[[1]]$address_components
$results[[1]]$address_components[[1]]

$results[[1]]$address_components[[1]]$long_name
[1] "119"
$results[[1]]$address_components[[1]]$short_name
[1] "119"
$results[[1]]$address_components[[1]]$types
$results[[1]]$address_components[[1]]$types[[1]]
[1] "street_number"

$results[[1]]$address_components[[2]]
$results[[1]]$address_components[[2]]$long_name
[1] "Kensington Church Street"
$results[[1]]$address_components[[2]]$short_name
[1] "Kensington Church St"
$results[[1]]$address_components[[2]]$types
$results[[1]]$address_components[[2]]$types[[1]]
[1] "route"

$results[[1]]$formatted_address
[1] "119 Kensington Church St, Kensington, London W8 7LN, UK"

$results[[1]]$geometry
$results[[1]]$geometry$location
$results[[1]]$geometry$location$lat
[1] 51.50691
$results[[1]]$geometry$location$lng
[1] -0.194801

$status
[1] "OK"
```

So, how does our function work? Firstly, `fromJSON` function from the jsonlite package is used to transform JSON output into R list. After that our function checks whether the API call was successful (*status = "OK"*), and if yes, it extracts from the list only the elements we need to construct the final data frame. It's a bit tricky to retrieve a *city* name, as firstly we need to find out under what sequence number it is stored inside the `address_components`. For *contacts* it's also important to replace all `NULL`, which might appear if Google has no information about the phone number or the website, with `NA`, so we get no error while generating the final data frame.

```
# ////////////////////////////////////////////////
# SUPLEMENTARY FUNCTIONS
# ////////////////////////////////////////////////

# ////////////////////////////////////////////////
# 3. EXTRACT DATA FROM JSON
# ////////////////////////////////////////////////

get_geodata_from_json_google <- function(geodata_json) {
    # load library
    library(jsonlite)

    # convert json output into r object
    geodata <- lapply(geodata_json, fromJSON,simplifyVector = FALSE)

    # extract coordinates, address and city name

    lat_lng_a <- data.frame(lat = NA, lng = NA, address = NA, city = NA)

    for (i in 1:length(geodata)) {

          if (geodata[[i]]$status=="OK") {

                # extract coordinates and address

                lat <- geodata[[i]]$results[[1]]$geometry$location$lat
                lng <- geodata[[i]]$results[[1]]$geometry$location$lng
                address <- geodata[[i]]$results[[1]]$formatted_address

                # find out how many elements there are in "address_components"
                n <- length(geodata[[i]]$results[[1]]$address_components)

                # extract city and country

                for (j in 1:n) {

                    # extract the type of the "address_components"
                    type <- geodata[[i]]$results[[1]]$address_components[[j]]$types[[1]]

                    # extract the city name

                    if (type == "postal_town") {
                        city <- geodata[[i]]$results[[1]]$address_components[[j]]$long_name
                    }

                }

                lat_lng_a[i, ] <- c(lat, lng, address, city)

          } else {
                lat_lng_a[i, ] <- NA
          }

    }

    return(lat_lng_a)

}

# ///////////////////////////////////////////////

get_place_id_from_json_google <- function(place_json) {
```

```r
    # load library
    library(jsonlite)

    # convert json output into r object
    place_search <- lapply(place_json, fromJSON,simplifyVector = FALSE)

    # extract place id

    place_id <- list()

    for (i in 1:length(place_search)) {

          if (place_search[[i]]$status=="OK") {
                place_id[[i]] <- place_search[[i]]$candidates[[1]]$place_id
          } else {
                place_id[[i]] <- NA
          }
    }

    return(place_id)

}

# ////////////////////////////////////////////////

get_contacts_from_json_google <- function(place_details_json) {

    # load library
    library(jsonlite)

    # convert json output into r object
    place_details <- lapply(place_details_json, fromJSON, simplifyVector = FALSE)

    # extract phone number and website

    contacts <- data.frame("phone number" = NA, "website" = NA)

    for (i in 1:length(place_details)) {

          if (place_details[[i]]$status=="OK") {

                # get data

                phone_number <- place_details[[i]]$result$formatted_phone_number
                website <- place_details[[i]]$result$website

                # get rid of NULLs

                info <- list(phone_number, website)

                for (j in 1:length(info)) {
                    if (is.null(info[[j]])) info[[j]] <- NA
                }
                # create output data frame
                contacts[i, ] <- info

          } else {
                contacts[i, ] <- NA
          }
    }

    return(contacts)

}

# ////////////////////////////////////////////////
```

0.7.2.4 Main Function We have already provided the description of our main function. Let's now just explain how it works.

Firstly, it calls `check_arguments` function to see if all the arguments user provided are in a correct format. If some of them are not, function stops execution and returns descriptive error message.

The next step is getting coordinates and address from the Google Maps. Firstly, function checks if user actually wants this information (i.e. *coordinates* and/or *address* are present in the `fields` argument) and if yes, it calls `url_google_geocoding` function to construct the API call and `getURL` function from the RCurl package to actually make it.

After we receive response from Google, we need to transform it from JSON format into R list by using `get_geodata_from_json_google` function. And once it's done, the result is stored in the `geodata_df` data frame.

Later, the very same procedure is repeated for contacts (i.e. *phone number* and *website*). And that's all.

Here is the code.

```
# ////////////////////////////////////////////////
# MAIN FUNCTION
# ////////////////////////////////////////////////

geocode_google <- function(search_query, fields = "coordinates", key) {

    # STOP RUNNING THE FUNCTION IF ARGUMENTS ARE INCORRECT

    errors <- check_arguments(search_query, fields, key)

    if (errors[[1]]) {
            stop(errors[[2]])
    }

    # LOAD LIBRARIES

    library(RCurl)

    # EXTRACT COORDINATES

        if (any(c("coordinates", "address") %in% fields) || "all" %in% fields) {
                # construct url for geocoding
                url_geocode <- url_google_geocoding(search_query, key)

                # get data from google
                geodata_json <- getURL(url_geocode)

                # get data from json output

                geodata_df <- as.data.frame(sapply(search_query, as.character),
                                            stringsAsFactors = FALSE)
                names(geodata_df) <- "search query"
                rownames(geodata_df) <- NULL
                geodata_df[, 2:5] <- get_geodata_from_json_google(geodata_json)

                # return dataframe with the geodata

                if (all(c("coordinates", "address") %in% fields) || "all" %in% fields) {
                    geodata_df
                } else if ("coordinates" %in% fields) {
                    geodata_df <- geodata_df[, 1:3]
                } else {
                    geodata_df <- geodata_df[, c(1, 4:5)]
                }

        }

    # EXTRACT CONTACTS

        if ("contacts" %in% fields || "all" %in% fields) {

            # /// get place_id from Place Search API ///

            # construct url for place search
            url_place_search <- url_google_place_search(search_query, key)

            # get data from google
            place_json <- getURL(url_place_search)

            # get place_id from json output
            place_id <- get_place_id_from_json_google(place_json)

            # /// get contacts from Place Details API ///

            # construct url for place details
            url_place_details <- url_google_place_details(place_id, key)

            # get data from google
            place_details_json <- getURL(url_place_details)

            # get place_id from json output
            contacts <- get_contacts_from_json_google(place_details_json)

            # /// add contacts to our output data frame ///
```

```
            if (!exists("geodata_df")) {
                geodata_df <- as.data.frame(sapply(search_query, as.character),
                                        stringsAsFactors = FALSE)
                names(geodata_df) <- "search query"
                rownames(geodata_df) <- NULL
            }

            geodata_df[, c("phone", "web page")] <- contacts

        }

    return(geodata_df)

}

# ////////////////////////////////////////////////
```

Now, we can finally call our function and check results.

```
# replace "api_key" with your API key
pubs_google <- geocode_google(pubs, "all", api_key)

# check results
pubs_google
```

```
##                               search query        lat         lng
## 1                    The Angel, Bermondsey 51.5006419 -0.0590456
## 2        The Churchill Arms, Notting Hill 51.5069117  -0.194801
## 3    The Auld Shillelagh, Stoke Newington 51.5620538 -0.0794564
## 4                 The Sekforde, Clerkenwell 51.5243685 -0.1044597
## 5                    The Dove, Hammersmith 51.4905047 -0.2348502
## 6  The Crown and Sugar Loaf, Fleet Street 51.5140163 -0.1049939
## 7                        The Lamb, Holborn 51.5230876 -0.1190473
## 8          Prince of Greenwich, Greenwich  51.476194  -0.009957
## 9            Ye Olde Mitre, Hatton Garden 51.5184089 -0.1073185
## 10                   The Glory, Haggerston 51.5362036 -0.0770533
## 11                    The Blue Posts, Soho  51.511127  -0.132611
## 12  The Old Bank of England, Fleet Street 51.5138962 -0.1114818
##                                                              address   city
## 1                            101 Bermondsey Wall E, London SE16 4NB, UK London
## 2              119 Kensington Church St, Kensington, London W8 7LN, UK London
## 3  105 Stoke Newington Church St, Stoke Newington, London N16 0UD, UK London
## 4                        34 Sekforde St, Farringdon, London EC1R 0HA, UK London
## 5                        19 Upper Mall, Hammersmith, London W6 9TA, UK London
## 6                                     98 Fleet St, London EC4Y 1DH, UK London
## 7                   94 Lamb's Conduit St, Holborn, London WC1N 3LZ, UK London
## 8                        72 Royal Hill, Greenwich, London SE10 8RT, UK London
## 9                       1 Ely Ct, Ely Pl, Holborn, London EC1N 6SJ, UK London
## 10                     281 Kingsland Rd, Haggerston, London E2 8AS, UK London
## 11                         28 Rupert St, West End, London W1D 6DJ, UK London
## 12                          194 Fleet St, Holborn, London EC4A 2LT, UK London
##             phone
## 1  020 7394 3214
## 2  020 7727 4242
## 3  020 7249 5951
## 4  020 7250 0010
## 5  020 8748 9474
## 6            <NA>
## 7  020 7405 0713
## 8   07940 596381
## 9  020 7405 4751
## 10 020 7684 0794
## 11  07921 336010
## 12 020 7430 2255
##
## 1                                                                     https://website--73151222016777
## 2  https://www.churchillarmskensington.co.uk/?utm_source=googlemybusiness&utm_medium=organic&utm_cam
## 3
## 4
## 5          https://www.dovehammersmith.co.uk/?utm_source=googlemybusiness&utm_medium=organic&utm_cam
## 6
## 7                             http://www.thelamblondon.com/?utm_source=local&utm_medium=organic&utm_ca
## 8                                                                                       https://www.t
## 9       https://www.yeoldemitreholborn.co.uk/?utm_source=googlemybusiness&utm_medium=organic&utm_cam
## 10                                                                                             https:/
## 11                                                                                                    h
## 12                                                                                               http:/
```

0.7.3 Reverse Geocoding Using Google Maps API

Below is pretty much the same function but for reverse geocoding. This time without checking arguments and returning contact details - just the address of the place based on its coordinates. I do not give here any detailed explanations as by this time you are well-equipped to understand the code on your own.

Apart from the `key` argument, the main function needs `coordinates` as an input, which can be either:

- a vector with latitude and longitude (a single request);
- a list of latitude/longitude vectors;
- a matrix with two columns - latitude and longitude;
- a data frame with two columns - latitude and longitude.

The function accepts as `coordinates` both numeric and string values.

Please note that Google might return a couple of results per each our call but we are using only the first one - `results[[1]]` - which corresponds to the best match in Google's opinion.

Also be careful with hard-coding references to the elements you want to extract from the R list. For example, in our case the 5th helement `$address_components[[5]]$long_name` might refer either to a city - London (`$address_components$types = "postal_town"`), level 2 administrative area - Greater London (`$address_components$types = "administrative_area_level_2"`) or level 1 administrative area - England (`$address_components$types = "administrative_area_level_1"`). So, in this case we have to loop through the R list to find the `types` of information we need and extract the corresponding `long_name`.

```
# ////////////////////////////////////////////////
# SUPLEMENTARY FUNCTIONS
# ////////////////////////////////////////////////
```

```r
# ///////////////////////////////////////////////
# 1. GENERATE API CALLS
# ///////////////////////////////////////////////

url_google_rev_geocoding <- function(coordinates_url, key_url) {

    # load libraries
    library(RCurl)

    # convert everything into data frame

    if (is.matrix(coordinates_url) || is.data.frame(coordinates_url)) {
        coordinates <- data.frame(matrix(NA, nrow(coordinates_url), ncol(coordinates_url)))
        names(coordinates) <- c("lat", "lng")
        coordinates[, 1] <- coordinates_url[, 1]
        coordinates[, 2] <- coordinates_url[, 2]
    } else if (is.list(coordinates_url)) {
        coordinates <- data.frame(matrix(NA, nrow = length(coordinates_url), ncol = 2))
        names(coordinates) <- c("lat", "lng")
        for (i in 1:length(coordinates_url)) {
            coordinates[i, 1] <- coordinates_url[[i]][1]
            coordinates[i, 2] <- coordinates_url[[i]][2]
        }
    } else if (is.vector(coordinates_url)) {
        coordinates <- data.frame(lat = NA, lng = NA)
        coordinates[1,1] <- coordinates_url[1]
        coordinates[1,2] <- coordinates_url[2]
    }

    coordinates$lat_lng <- paste0(coordinates$lat, ",", coordinates$lng)

    # google gecoding api url
    url_geocoding_api <- "https://maps.googleapis.com/maps/api/geocode/"

    # construct search request for reverse geocoding
    url_rev_geocoding_call <- paste0(url_geocoding_api, "json",
                             "?latlng=", coordinates$lat_lng, "&key=", key_url)

    # return data frame with coordinates and API call

    coordinates$api_call <- url_rev_geocoding_call

    return(coordinates)

}

# ///////////////////////////////////////////////
# 2. EXTRACT DATA FROM JSON
# ///////////////////////////////////////////////

get_rev_geodata_from_json_google <- function(geodata_json) {

    # load library
    library(jsonlite)

    # convert json output into r object
    geodata <- lapply(geodata_json, fromJSON,simplifyVector = FALSE)

    # extract address, city and country from the json output

    address_df <- data.frame(address = NA, city = NA, country = NA)

    for (i in 1:length(geodata)) {

          if (geodata[[i]]$status=="OK") {

                # extract address
                address <- geodata[[i]]$results[[1]]$formatted_address

                # find out how many elements there are in "address_components"
                n <- length(geodata[[i]]$results[[1]]$address_components)

                # extract city and country

                for (j in 1:n) {

                      # extract type of "address_components"
                      type <- geodata[[i]]$results[[1]]$address_components[[j]]$types[[1]]

                      # extract city and country

                      if (type == "postal_town") {
                            city <- geodata[[i]]$results[[1]]$address_components[[j]]$long_name
                      } else if (type == "country") {
                            country <- geodata[[i]]$results[[1]]$address_components[[j]]$long_name
                      }

                }

                # prepare output
                address_df[i, ] <- c(address, city, country)

          } else {
                address_df[i, ] <- NA
          }

    }

    return(address_df)

}

# ////////////////////////////////////////////////
# MAIN FUNCTION
# ////////////////////////////////////////////////
```

```r
rev_geocode_google <- function(coordinates, key) {

      # load libraries
      library(RCurl)

      # construct url for reverse geocoding
      rev_geocoding_info <- url_google_rev_geocoding(coordinates, key)

      # get data from google
      geodata_json <- getURL(rev_geocoding_info$api_call)

      # get data from json output
      geodata_df <- rev_geocoding_info[, c("lat", "lng")]
      geodata_df[, 3:5] <- get_rev_geodata_from_json_google(geodata_json)

      # return dataframe with the geodata
      return(geodata_df)

}

# ////////////////////////////////////////////////
```

Below are the results of running this function on the sample of London pubs whose coordinates we got earlier from the same API.

```r
# extract coordinates from pubs_google
pubs_google_crd <- pubs_google[ , c("lat", "lng")]

# replace "api_key" with your API key
pubs_rev_google <- rev_geocode_google(pubs_google_crd, api_key)

# check results
pubs_rev_google <- cbind(pubs_df, pubs_rev_google)
pubs_rev_google
```

```
##                                      Pubs        lat         lng
## 1                   The Angel, Bermondsey 51.5006419 -0.0590456
## 2         The Churchill Arms, Notting Hill 51.5069117  -0.194801
## 3    The Auld Shillelagh, Stoke Newington 51.5620538 -0.0794564
## 4                 The Sekforde, Clerkenwell 51.5243685 -0.1044597
## 5                   The Dove, Hammersmith 51.4905047 -0.2348502
## 6  The Crown and Sugar Loaf, Fleet Street 51.5140163 -0.1049939
## 7                       The Lamb, Holborn 51.5230876 -0.1190473
## 8          Prince of Greenwich, Greenwich  51.476194  -0.009957
## 9            Ye Olde Mitre, Hatton Garden 51.5184089 -0.1073185
## 10                 The Glory, Haggerston 51.5362036 -0.0770533
## 11                   The Blue Posts, Soho  51.511127  -0.132611
## 12  The Old Bank of England, Fleet Street 51.5138962 -0.1114818
##                                                             address   city
## 1                              101 Bermondsey Wall E, London SE16 4NB, UK London
## 2              119 Kensington Church St, Kensington, London W8 7LN, UK London
## 3 105 Stoke Newington Church St, Stoke Newington, London N16 0UD, UK London
## 4                           34 Sekforde St, Farringdon, London EC1R 0HA, UK London
## 5                        19 Upper Mall, Hammersmith, London W6 9TA, UK London
## 6                                     98 Fleet St, London EC4Y 1DH, UK London
## 7                   94 Lamb's Conduit St, Holborn, London WC1N 3LZ, UK London
## 8         Prince Albert, 72 Royal Hill, Greenwich, London SE10 8RT, UK London
## 9                          1 Ely Ct, Ely Pl, Holborn, London EC1N 6SJ, UK London
## 10                   281A Kingsland Rd, Haggerston, London E2 8AS, UK London
## 11                             28 Rupert St, West End, London W1D 6DJ, UK London
## 12                             194 Fleet St, Holborn, London EC4A 2LT, UK London
##           country
## 1  United Kingdom
## 2  United Kingdom
## 3  United Kingdom
## 4  United Kingdom
## 5  United Kingdom
## 6  United Kingdom
## 7  United Kingdom
## 8  United Kingdom
## 9  United Kingdom
## 10 United Kingdom
## 11 United Kingdom
## 12 United Kingdom
```

0.7.4 Nominatim API

Now let's turn our attention to the OSM's Nominatim API.

The Nominatim search API allows you to look for a specific location based on its description or address. It supports both structured and free-text requests. The search query may also contain special phrases which correspond to the specific OpenStreetMap tags. In our case this special phrase is a *"pub"*.

The reverse geocoding API generates an address from a place's latitude and longitude.

The formats of API calls are presented below.

```
# geocoding format
https://nominatim.openstreetmap.org/search/<query>?<params>

# geocoding example
https://nominatim.openstreetmap.org/search/The%20Churchill%20Arms,%20Notting%20Hill?format=json&polygon=

# reverse geocoding format
https://nominatim.openstreetmap.org/reverse?<query>

# reverse geocoding example
https://nominatim.openstreetmap.org/reverse?format=json&lat=51.5068722&lon=-0.1948221&zoom=18&addressde
```

Some parameters are common for both geocoding and reverse geocoding calls:

- `format` = [html | xml | json | jsonv2 | geojson | geocodejson] - output format;
- `addressdetails` = [0|1] - include breakdown of address into elements;
- `extratags` = [0|1] - additional information (wiki page, opening hours etc.);
- `accept-language` - in what language to display the search results (English = en);
- `email` - unless you provide an email address, which allows to track your activity, you will not be able to use the API in your app (error message would appear).

Some parameters are peculiar for each API:

- **search**:
  - `query` - a free text or an address;
  - `countrycodes` - limit the search by ISO 3166-1 alpha-2 country codes (the UK = gb);
  - `limit` - limit the number of returned results;
- **reverse**:
  - `query` = lat, lon - in WGS 84 format;
  - `namedetails` = [0|1] - include a list of alternative names in the results;
  - `zoom` = [0-18] - level of detail required for the address (default is 18, i.e. a specific building).

The `query`, `format` and `email` are required parameters, while the rest are optional. We will not be using `namedetails` parameter in our function and will not change the value of `zoom` parameter - I provided them just for your reference.

One important aspect here is the usage of tags, which point out to a specific piece of information provided by OpenStreeMap mappers. Some of these tags have duplicates (like email and phone and website vs similar tags in contact namespace), so different people might label the same sort of information with different tags and you need to account for that in your app.

There are also a few requirements, which you must respect to be able to use Nominatim service:

- restriction on the number of requests sent by the same website/app - one request per second per app;
- bulk geocoding of large amounts of data is discouraged but smaller one-time tasks (our case) are allowed;
- search results must be cached, so not to send the same request more than once.

0.7.5 Geocoding Using Nominatim API

The function below replicates the one we have built for Google Maps API, so we will not be describing it in detail.

The only significant difference is that we added two additional optional arguments: `country`, which corresponds to the `countrycodes` parameter of API call and is used to restrict your search to some counties only (by default it is not used) and `language` that corresponds to the `accept-language` parameter and allows you to choose the language in which to display results (default is English). Both arguments need to be provided in the format of a string: `country` as a comma-delimited list of codes (e.g. "gb,dr,fr") and `language` as a single value (e.g. "es").

```r
# ////////////////////////////////////////////////
# SUPLEMENTARY FUNCTIONS
# ////////////////////////////////////////////////

# ////////////////////////////////////////////////
# 1. GENERATE API CALLS
# ////////////////////////////////////////////////

url_nominatim_search <- function(search_query_url, country_url,
                                 language_url, email_url) {

    # load libraries
    library(RCurl)

    # nominatim search api url
    url_nominatim_search_api <- "https://nominatim.openstreetmap.org/search/"

    # convert input into a list
    search_query_url <- sapply(search_query_url, as.list)

    # percent-encode search request
    search_query_url <- sapply(search_query_url, URLencode)

    # parameters

    if (!is.null(country_url)) {
          country_url <- paste0("&countrycodes=", country_url)
    }

    parameters_url <- paste0("?format=json",
                             "&addressdetails=1&extratags=1&limit=1",
                             country_url, "&accept-language=", language_url,
                             "&email=", email_url)

    # construct search request for geocode
    url_nominatim_search_call <- paste0(url_nominatim_search_api,
                                        search_query_url, parameters_url)

    return(url_nominatim_search_call)

}

# ////////////////////////////////////////////////
# 2. EXTRACT DATA FROM JSON
# ////////////////////////////////////////////////

get_geodata_from_json_nominatim <- function(geodata_json) {

    # load library
    library(jsonlite)

    # convert json output into r object
    geodata <- lapply(geodata_json, fromJSON,simplifyVector = FALSE)

    # extract coordinates, address and contacts

    lat_lng_a_c <- data.frame(lat = NA, lng = NA, address = NA, pub_name = NA,
                              street_name = NA, house_number = NA, suburb = NA,
                              postcode = NA, state_district = NA, website_1 = NA,
                              website_2 = NA, website_3 = NA, phone_1 = NA,
                              phone_2 = NA, email_1 = NA, email_2 = NA)

    for(i in 1:length(geodata)) {

          if(length(geodata[[i]]) != 0) {
                # get data

                lat <- geodata[[i]][[1]]$lat
                lng <- geodata[[i]][[1]]$lon
                address <- geodata[[i]][[1]]$display_name
                pub_name <- geodata[[i]][[1]]$address$pub
                street_name <- geodata[[i]][[1]]$address$road
                house_number <- geodata[[i]][[1]]$address$house_number
                suburb <- geodata[[i]][[1]]$address$suburb
                postcode <- geodata[[i]][[1]]$address$postcode
                state_district <- geodata[[i]][[1]]$address$state_district
                website_1 <- geodata[[i]][[1]]$extratags$website
                website_2 <- geodata[[i]][[1]]$extratags$url
                website_3 <- geodata[[i]][[1]]$extratags$`contact:website`
                phone_1 <- geodata[[i]][[1]]$extratags$phone
                phone_2 <- geodata[[i]][[1]]$extratags$`contact:phone`
                email_1 <- geodata[[i]][[1]]$extratags$email
                email_2 <- geodata[[i]][[1]]$extratags$`contact:website`

                # get rid of NULLs

                info <- list(lat, lng, address, pub_name, street_name,
                             house_number, suburb, postcode, state_district,
                             website_1, website_2, website_3,
                             phone_1, phone_2, email_1, email_2)

                for (j in 1:length(info)) {
                    if (is.null(info[[j]])) info[[j]] <- NA
                }

                # create output data frame

                lat_lng_a_c[i, ] <- info

          } else {
                lat_lng_a_c[i, ] <- NA
          }
    }

    return(lat_lng_a_c)

}

# //////////////////////////////////////////////////
# MAIN FUNCTION
# //////////////////////////////////////////////////

geocode_nominatim <- function(search_query, country = NULL, language = "en",
                              fields = "coordinates", email) {

    # LOAD LIBRARIES

    library(RCurl)
    # EXTRACT DATA

        # construct url for geocoding
        url_geocode <- url_nominatim_search(search_query, country, language, email)

        # get data from nominatim
        # wait 3 seconds between each call

        geodata_json <- list()

        for (i in 1:length(url_geocode)) {
            geodata_json[i] <- getURL(url_geocode[i])
            Sys.sleep(3)
        }

        # get data from json output

        geodata_df <- as.data.frame(sapply(search_query, as.character),
                                    stringsAsFactors = FALSE)
        names(geodata_df) <- "search query"
        rownames(geodata_df) <- NULL

        geodata_df[, 2:17] <- get_geodata_from_json_nominatim(geodata_json)
        geodata_df_query <- data.frame(search_query = geodata_df[, 1],
                                       stringsAsFactors = FALSE)
        geodata_df_coordinates <- geodata_df[, 2:3]
        geodata_df_address <- geodata_df[, 4:10]
        geodata_df_contacts <- geodata_df[, 11:17]

        # return dataframe with the geodata

        geodata_result <- geodata_df_query

        if("all" %in% fields) {
            geodata_result <- cbind(geodata_result, geodata_df[, 2:17])
        }

        if("coordinates" %in% fields) {
            geodata_result <- cbind(geodata_result, geodata_df_coordinates)
        }

        if("address" %in% fields) {
            geodata_result <- cbind(geodata_result, geodata_df_address)
        }

        if("contacts" %in% fields) {
            geodata_result <- cbind(geodata_result, geodata_df_contacts)
        }

    return(geodata_result)

}
```

```
# ////////////////////////////////////////////////
```

Let's see the results from running this function.

```
# replace "email" with your email address

pubs_nominatim <- geocode_nominatim(pubs_m, country = "gb", fields = "all", email = email)

# let's now see the results

pubs_nominatim[, c(1:4)]
pubs_nominatim[, c(1, 5:10)]
pubs_nominatim[, c(1, 11:13)]
pubs_nominatim[, c(1, 14:17)]
```

```
##                               search_query                lat
## 1                    The Angel, Bermondsey        51.50063675
## 2        The Churchill Arms, Notting Hill 51.506872200000004
## 3   The Auld Shillelagh, Stoke Newington         51.5620186
## 4                the Sekforde, Clerkenwell         51.5242876
## 5                   The Dove, Hammersmith 51.490508500000004
## 6                 The Crown and Sugar Loaf         51.5140324
## 7                      The Lamb, Holborn         51.5230437
## 8          Prince of Greenwich, Greenwich         51.4762278
## 9                             Ye Olde Mitre         51.5184299
## 10                   The Glory, Haggerston               <NA>
## 11                    The Blue Posts, Soho         51.5128957
## 12 The Old Bank of England, Fleet Street         51.5139117
##                    lng
## 1  -0.05906114651162744
## 2  -0.19482219080740487
## 3            -0.0794532
## 4  -0.10442487273234387
## 5  -0.23485741509400304
## 6            -0.1050005
## 7  -0.11903102762686923
## 8            -0.0098418
## 9  -0.10738277040280805
## 10                <NA>
## 11 -0.13943685289494165
## 12 -0.11146836718248274
##
## 1                       The Angel, 101, Bermondsey Wall East, Bermondsey Spa, Bermondsey, London Borou
## 2                             The Churchill Arms, Campden Street, Notting Hill, Royal Borough of Kensi
## 3  The Auld Shillelagh, 105, Stoke Newington Church Street, South Hornsey, Stoke Newington, London Bo
## 4                                    The Sekforde, Woodbridge Street, Angel, Clerkenwell, London Borou
## 5                                       The Dove, 19, Upper Mall, Brook Green, London Borough of Hamme
## 6                                                                Crown & Sugar Loaf, 26, Bride Lane
## 7                                  The Lamb, 94, Lamb's Conduit Street, Holborn, Bloomsbury, London Bo
## 8                                                 The Prince Of Greenwich, 72, Royal Hill, Royal Borou
## 9                                                         Ye Olde Mitre, Ely Place, Holborn, London Bo
## 10
## 11                                                            The Blue Posts, Kingly Street, Soho, Ci
## 12                                              The Old Bank of England, 194, Fleet Street

##                             search_query                pub_name
## 1                  The Angel, Bermondsey               The Angel
## 2        The Churchill Arms, Notting Hill      The Churchill Arms
## 3   The Auld Shillelagh, Stoke Newington     The Auld Shillelagh
## 4              The Sekforde, Clerkenwell            The Sekforde
## 5                  The Dove, Hammersmith                The Dove
## 6               The Crown and Sugar Loaf      Crown & Sugar Loaf
## 7                     The Lamb, Holborn                The Lamb
## 8          Prince of Greenwich, Greenwich The Prince Of Greenwich
## 9                          Ye Olde Mitre           Ye Olde Mitre
## 10                 The Glory, Haggerston                    <NA>
## 11                  The Blue Posts, Soho          The Blue Posts
## 12 The Old Bank of England, Fleet Street The Old Bank of England
##                      street_name house_number          suburb postcode
## 1           Bermondsey Wall East          101      Bermondsey SE16 4TU
## 2                 Campden Street         <NA>    Notting Hill   W8 7EL
## 3  Stoke Newington Church Street          105 Stoke Newington  N16 0UD
## 4              Woodbridge Street         <NA>     Clerkenwell EC1R 0DG
## 5                     Upper Mall           19     Brook Green   W6 9TA
## 6                     Bride Lane           26          Temple EC4Y 8DT
## 7           Lamb's Conduit Street          94      Bloomsbury WC1N 3LZ
## 8                     Royal Hill           72            <NA> SE10 8RT
## 9                      Ely Place         <NA>            <NA> EC1N 6SJ
## 10                          <NA>         <NA>            <NA>     <NA>
## 11                          <NA>         <NA>            Soho  W1F 7PA
## 12                  Fleet Street          194          Temple EC4A 2LT
##    state_district
## 1  Greater London
## 2  Greater London
## 3  Greater London
## 4  Greater London
## 5  Greater London
## 6  Greater London
## 7  Greater London
## 8  Greater London
## 9  Greater London
## 10           <NA>
## 11 Greater London
## 12 Greater London

##                             search_query
## 1                  The Angel, Bermondsey
## 2        The Churchill Arms, Notting Hill
## 3   The Auld Shillelagh, Stoke Newington
## 4              The Sekforde, Clerkenwell
## 5                  The Dove, Hammersmith
## 6               The Crown and Sugar Loaf
## 7                     The Lamb, Holborn
## 8          Prince of Greenwich, Greenwich
## 9                          Ye Olde Mitre
## 10                 The Glory, Haggerston
## 11                  The Blue Posts, Soho
## 12 The Old Bank of England, Fleet Street
##                                                    website_1 website_2
## 1                                                       <NA>        NA
## 2              https://www.churchillarmskensington.co.uk/        NA
## 3                                                       <NA>        NA
## 4                                                       <NA>        NA
## 5                      https://www.dovehammersmith.co.uk/        NA
## 6                                                       <NA>        NA
## 7       https://www.youngs.co.uk/pub-detail.asp?PubID=421        NA
## 8                https://www.theprinceofgreenwichpub.com        NA
## 9                                                       <NA>        NA
## 10                                                      <NA>        NA
## 11 http://traditionalpubslondon.co.uk/blueposts/index.php        NA
## 12                                                      <NA>        NA
##                           website_3
## 1                              <NA>
## 2                              <NA>
## 3                              <NA>
## 4                              <NA>
## 5                              <NA>
## 6                              <NA>
## 7                              <NA>
## 8                              <NA>
## 9                              <NA>
## 10                             <NA>
## 11                             <NA>
## 12 http://www.oldbankofengland.co.uk/
```

```
##                            search_query          phone_1 phone_2
## 1                 The Angel, Bermondsey             <NA>      NA
## 2        The Churchill Arms, Notting Hill           <NA>      NA
## 3   The Auld Shillelagh, Stoke Newington            <NA>      NA
## 4              The Sekforde, Clerkenwell            <NA>      NA
## 5                 The Dove, Hammersmith             <NA>      NA
## 6               The Crown and Sugar Loaf            <NA>      NA
## 7                     The Lamb, Holborn             <NA>      NA
## 8          Prince of Greenwich, Greenwich +44 20 8692 6089      NA
## 9                          Ye Olde Mitre            <NA>      NA
## 10                 The Glory, Haggerston            <NA>      NA
## 11                  The Blue Posts, Soho            <NA>      NA
## 12 The Old Bank of England, Fleet Street   +44 2074302255      NA
##                             email_1                              email_2
## 1                              <NA>                                 <NA>
## 2                              <NA>                                 <NA>
## 3                              <NA>                                 <NA>
## 4                              <NA>                                 <NA>
## 5                              <NA>                                 <NA>
## 6                              <NA>                                 <NA>
## 7                              <NA>                                 <NA>
## 8  theprinceofgreenwichpub@gmail.com                                <NA>
## 9                              <NA>                                 <NA>
## 10                             <NA>                                 <NA>
## 11                             <NA>                                 <NA>
## 12                             <NA> http://www.oldbankofengland.co.uk/
```

0.7.6 Reverse Geocoding Using Nominatim API

Similarly, the reverse geocoding function below to a large extent resembles the one we have built for the Google Maps API.

```r
# ////////////////////////////////////////////////
# SUPLEMENTARY FUNCTIONS
# ////////////////////////////////////////////////

# ////////////////////////////////////////////////
# 1. GENERATE API CALLS
# ////////////////////////////////////////////////

url_nominatim_rev_geocoding <- function(coordinates_url, language_url, email_url) {

    # load libraries
    library(RCurl)

    # convert everything into a data frame

    if (is.matrix(coordinates_url) || is.data.frame(coordinates_url)) {
        coordinates <- data.frame(matrix(NA, nrow(coordinates_url), ncol(coordinates_url)))
        names(coordinates) <- c("lat", "lng")
        coordinates[, 1] <- coordinates_url[, 1]
        coordinates[, 2] <- coordinates_url[, 2]
    } else if (is.list(coordinates_url)) {
        coordinates <- data.frame(matrix(NA, nrow = length(coordinates_url), ncol = 2))
        names(coordinates) <- c("lat", "lng")
        for (i in 1:length(coordinates_url)) {
            coordinates[i, 1] <- coordinates_url[[i]][1]
            coordinates[i, 2] <- coordinates_url[[i]][2]
        }
    } else if (is.vector(coordinates_url)) {
        coordinates <- data.frame(lat = NA, lng = NA)
        coordinates[1,1] <- coordinates_url[1]
        coordinates[1,2] <- coordinates_url[2]
    }

    # nominatim reverse api url
    url_nominatim_reverse_api <- "https://nominatim.openstreetmap.org/reverse"

    # parameters

    lat <- coordinates$lat
    lon <- coordinates$lng

    parameters_url <- paste0("?format=json", "&lat=", lat, "&lon=", lon,
                         "&addressdetails=1&extratags=1","&accept-language=",
                         language_url, "&zoom=18", "&email=", email_url)

    # construct search request for geocode
    url_nominatim_reverse_call <- paste0(url_nominatim_reverse_api, parameters_url)

    # return data frame with coordinates and API call
    coordinates$api_call <- url_nominatim_reverse_call

    return(coordinates)

}

# ///////////////////////////////////////////////
# 2. EXTRACT DATA FROM JSON
# ///////////////////////////////////////////////

get_rev_geodata_from_json_nominatim <- function(geodata_json) {

    # load library
    library(jsonlite)

    # convert json output into r object
    geodata <- lapply(geodata_json, fromJSON,simplifyVector = FALSE)

    # extract address, city and country

    address_df <- data.frame(address = NA, pub_name = NA, street_name = NA,
                             house_number = NA, suburb = NA, postcode = NA,
                             state_district = NA, country = NA)

    for(i in 1:length(geodata)) {

        if(length(geodata[[i]]) != 0) {

            # get data

            address <- geodata[[i]]$display_name
            pub_name <- geodata[[i]]$address$pub
            street_name <- geodata[[i]]$address$road
            house_number <- geodata[[i]]$address$house_number
            suburb <- geodata[[i]]$address$suburb
            postcode <- geodata[[i]]$address$postcode
            state_district <- geodata[[i]]$address$state_district
            country <- geodata[[i]]$address$country

            # get rid of NULLs

            info <- list(address, pub_name, street_name, house_number,
                         suburb, postcode, state_district, country)

            for (j in 1:length(info)) {
                if (is.null(info[[j]])) info[[j]] <- NA
            }

            # create output data frame

            address_df[i, ] <- info

        } else {
            address_df[i, ] <- NA
        }
    }

    return(address_df)

}

# ///////////////////////////////////////////////
# MAIN FUNCTION
# ///////////////////////////////////////////////

rev_geocode_nominatim <- function(coordinates, language = "en", email) {

      # load libraries
      library(RCurl)

      # construct url for reverse geocoding
      rev_geocoding_info <- url_nominatim_rev_geocoding(coordinates, language, email)

      # get data from nominatim
      # wait 3 seconds between each call

      geodata_json <- list()

      for (i in 1:dim(rev_geocoding_info)[1]) {
          geodata_json[i] <- getURL(rev_geocoding_info$api_call[i])
          Sys.sleep(3)
      }

      # get data from json output
      geodata_df <- rev_geocoding_info[, c("lat", "lng")]
      geodata_df[, 3:10] <- get_rev_geodata_from_json_nominatim(geodata_json)

      # return dataframe with the geodata
      return(geodata_df)

}

# ///////////////////////////////////////////////
```

Here are the results from running this function on the sample of London pubs.

```
# extract coordinates from geocoding results

pubs_nominatim_crd <- pubs_nominatim[, c("lat", "lng")]

# replace "email" with your email address

pubs_rev_nominatim <- rev_geocode_nominatim(pubs_nominatim_crd, email = email)
pubs_rev_nominatim <- cbind(pubs_m_df, pubs_rev_nominatim)

# let's now see the results
```

```r
pubs_rev_nominatim[, 1:4]
pubs_rev_nominatim[, c(1, 5:11)]
```

```
##                                     Pubs                lat
## 1                  The Angel, Bermondsey        51.50063675
## 2       The Churchill Arms, Notting Hill 51.506872200000004
## 3    The Auld Shillelagh, Stoke Newington         51.5620186
## 4               The Sekforde, Clerkenwell         51.5242876
## 5                  The Dove, Hammersmith 51.490508500000004
## 6                The Crown and Sugar Loaf         51.5140324
## 7                     The Lamb, Holborn         51.5230437
## 8          Prince of Greenwich, Greenwich         51.4762278
## 9                         Ye Olde Mitre         51.5184299
## 10                  The Glory, Haggerston               <NA>
## 11                   The Blue Posts, Soho         51.5128957
## 12 The Old Bank of England, Fleet Street         51.5139117
##                    lng
## 1  -0.05906114651162744
## 2  -0.19482219080740487
## 3            -0.0794532
## 4  -0.10442487273234387
## 5  -0.23485741509400304
## 6            -0.1050005
## 7  -0.11903102762686923
## 8            -0.0098418
## 9  -0.10738277040280805
## 10                 <NA>
## 11 -0.13943685289494165
## 12 -0.11146836718248274
##
## 1                     The Angel, 101, Bermondsey Wall East, Bermondsey Spa, Bermondsey, London Borou
## 2                           The Churchill Arms, Campden Street, Notting Hill, Royal Borough of Kensi
## 3  The Auld Shillelagh, 105, Stoke Newington Church Street, South Hornsey, Stoke Newington, London Bo
## 4                                  The Sekforde, Woodbridge Street, Angel, Clerkenwell, London Borou
## 5                                     The Dove, 19, Upper Mall, Brook Green, London Borough of Hamme
## 6                                                                      Punch Tavern, 30, Bride Lane
## 7                                The Lamb, 94, Lamb's Conduit Street, Holborn, Bloomsbury, London Bo
## 8                                               The Prince Of Greenwich, 72, Royal Hill, Royal Borou
## 9                                                       Ye Olde Mitre, Ely Place, Holborn, London Bo
## 10
## 11                                                         The Blue Posts, Kingly Street, Soho, Ci
## 12                                                      The Old Bank of England, 194, Fleet Street

##                                     Pubs                pub_name
## 1                  The Angel, Bermondsey               The Angel
## 2       The Churchill Arms, Notting Hill      The Churchill Arms
## 3    The Auld Shillelagh, Stoke Newington     The Auld Shillelagh
## 4               The Sekforde, Clerkenwell            The Sekforde
## 5                  The Dove, Hammersmith                The Dove
## 6                The Crown and Sugar Loaf            Punch Tavern
## 7                     The Lamb, Holborn                The Lamb
## 8          Prince of Greenwich, Greenwich The Prince Of Greenwich
## 9                         Ye Olde Mitre           Ye Olde Mitre
## 10                The Glory, Haggerston                    <NA>
## 11                 The Blue Posts, Soho          The Blue Posts
## 12 The Old Bank of England, Fleet Street The Old Bank of England
##                        street_name house_number          suburb postcode
## 1            Bermondsey Wall East          101      Bermondsey SE16 4TU
## 2                  Campden Street         <NA>    Notting Hill   W8 7EL
## 3  Stoke Newington Church Street          105 Stoke Newington  N16 0UD
## 4               Woodbridge Street         <NA>     Clerkenwell EC1R 0DG
## 5                      Upper Mall           19     Brook Green   W6 9TA
## 6                      Bride Lane           30          Temple EC4Y 8DX
## 7            Lamb's Conduit Street           94      Bloomsbury WC1N 3LZ
## 8                      Royal Hill           72            <NA> SE10 8RT
## 9                       Ely Place         <NA>            <NA> EC1N 6SJ
## 10                           <NA>         <NA>            <NA>     <NA>
## 11                           <NA>         <NA>            Soho  W1F 7PA
## 12                   Fleet Street          194          Temple EC4A 2LT
##    state_district        country
## 1  Greater London United Kingdom
## 2  Greater London United Kingdom
## 3  Greater London United Kingdom
## 4  Greater London United Kingdom
## 5  Greater London United Kingdom
## 6  Greater London United Kingdom
## 7  Greater London United Kingdom
## 8  Greater London United Kingdom
## 9  Greater London United Kingdom
## 10           <NA>           <NA>
## 11 Greater London United Kingdom
## 12 Greater London United Kingdom
```

0.8 Building a Map With Leaflet Library

The common truth is that a genuine interest in the subject can only be drawn when training material is complemented with examples of practical application. I promised you that based on information we got from API we would build an interactive map and I intend to deliver on that promise.

One of the ways to easily build a map is using JavaScript Leaflet library. Leaflet is described on its website as: *"[...] the leading open-source JavaScript library for mobile-friendly interactive maps."* Many big tech companies, some media and even government bodies are using it: GitHub, Facebook, Pinterest, Financial Times, The Washington Post, Data.gov, European Commission are among the few. In our case, we will rely on the leaflet package from the RStudio, which makes it easy to integrate and control Leaflet maps in R. For the full documentation check the package's description on CRAN.

I will not describe here all the features this great tool provides because it's the topic for another full article. Rather let's concentrate on the most essential ones.

So, the process of creating a map in Leaflet involves three basic steps:

1. Create a map widget.
2. Add layers to your map.
3. Display the map.

Widget is essentially a backbone or a container for your map.

Layers allow you to add to the map such elements as:

- tiles - essentially the "skin" of your map, which defines its appearance and a level of detail. More about tiled maps;
- markers - can be used to show a particular location on a map;
- pop-ups and labels - can be used to add labels to your map. For example, to show an address or a contact information associated with some location;
- polygons - a specific region or area. For example, a district within a state;
- legends etc.

For our map we will be using a tile from OpenStreetMap (the default one for Leaflet) and plot the pubs' location based on the coordinates we extracted from the Nominatim. Additionally, we will add to the markers pop-ups with information about the pub's name, address and contact details. As Nominatim did not return full details about each pub, I searched this information on my own. We are not using any polygons or legends in our visualization, I added links just for your reference.

So, before we move on let's do some data preparation.

```
# copy the data from Nominatim API results
pubs_map <- pubs_nominatim

# add city
pubs_map$city = "London"

# add details about The Glory pub
pubs_map[10, 2:11] <- c("51.536327", "-0.077021", "The Glory, 281 Kingsland Rd, Haggerston, London, Grea

# ///////////////////////////
# CLEANSING OF CONTACT DETAILS
# ///////////////////////////

# remove emails
pubs_map <- pubs_map[, -c(16, 17)]

# clean phone number
pubs_map$phone <- NA
for (i in 1:dim(pubs_map)[1]) {
      pubs_phone <- pubs_map[i, c("phone_1", "phone_2")]
      match <- !is.na(pubs_phone)
      if (any(match)) {
            pubs_map[i, "phone"] <- pubs_phone[match][1]
      }
}

# remove unnecessary phone numbers
pubs_map <- pubs_map[, -c(14:15)]

# clean website
pubs_map$website <- NA
for (i in 1:dim(pubs_map)[1]) {
      pubs_website <- pubs_map[i, c("website_1", "website_2", "website_3")]
      match <- !is.na(pubs_website)
      if (any(match)) {
            pubs_map[i, "website"] <- pubs_website[match][1]
      }
}
```

```r
# remove unnecessary websites
pubs_map <- pubs_map[, -c(11:13)]

# //////////////////
# ADD MISSING DETAILS
# //////////////////

# street name
pubs_map$street_name[11] <- "Rupert Street"

# house number
pubs_map$house_number[c(2,4,9,11)] <- c("119", "34", "1", "28")

# suburb
pubs_map$suburb[c(8,9)] <- c("Greenwich", "Holborn")

# phone
# for #9 phone number is different from the one Nominatim provided
# for #12 phone number is corrent but I want the phone number format to be unified
pubs_map$phone <- c("+44 20 7394 3214", "+44 20 7727 4242", "+44 20 7249 5951", "+44 20 7250 0010", "+4

# website
# again for some pubs Nominatim provided a broken link
pubs_map$website[c(1,3,4,6,7,9,10,11)] <- c("https://website--7315122201677780705144-pub.business.site"

# create address to be displayed on the pop-up
cols <- c("house_number", "street_name", "suburb", "city", "postcode")
pubs_map$address_display <- do.call(paste, c(pubs_map[cols], sep=", "))

# change lat-lng data type
pubs_map$lat <- as.numeric(pubs_map$lat)
pubs_map$lng <- as.numeric(pubs_map$lng)
```

Now we can proceed with building the map itself.

Firstly, let's prepare the text to be displayed in the pop-up messages: pub's name, address and phone number. Website will not be showed separately but added as a hyperlink to the pub's name. We will use some html to render our text in the format we want. Here is one hint. Pop-up messages are displayed only when you click on the objects they are attached to. If you want to display some information when cursor is hovered over the marker, you need to use labels. However, unlike pop-ups, labels do not automatically recognize HTML syntax - you would need to use `HTML` function from the htmltools package to transform your message first. Once it's done we can "draw" our map.

Leaflet functions are quite self-explanatory. The only thing that might not be familiar to you is the pipe operator %>%, which was brought in by the tidyverse packages collection. Basically, it allows you to easily chain function calls by passing an output of one function as an argument of another. More information on that here.

Let's finally see the result.

```r
# text to be diplayed on pop-ups

website <- paste0("<a href='", pubs_map$website, "'>", pubs_map$pub_name, "</a>")
center <- "<div style='text-align:center'>"
```

```
name <- paste0(center, "<b>", website, "</b>", "</div>")
address <- paste0(center, pubs_map$address_display, "</div>")
phone <- paste0(center, pubs_map$phone, "</div>")

# building the map

pubs_map %>%
    leaflet() %>%
    addTiles() %>%
    addMarkers(~lng, ~lat, popup = paste0(name, address, phone))
```

0.9 Conclusion

In this tutorial we covered different methods of retrieving geocoding data using Google Maps and Nominatim APIs and showed how this data can be used to plot particular locations on a map using JavaScript Leaflet library. I hope this guide will serve you as a starting point for exploring all the different kinds of APIs and mapping tools.