Automatic Sampling and Analysis of YouTubeData

Collecting Data With the tuber Package for R

Julian Kohne
Johannes Breuer
M. Rohangis Mohseni

2022-02-21

Collecting *YouTube* Data with R

As stated in the Introduction, there are several packages for collecting *YouTube* data in R: `tuber`, `vosonSML`, `VOSONDash`, and `youtubecaption`.

In this session, we will focus on working with the `tuber` package. While `vosonSML` and `VOSONDash` are very powerful for generating (and analyzing) network data (and can also collect more comments per video as we will see later), `tuber` offers functions for collecting different types of data about channels, playlists, and videos.

The `tuber` Package

With the tuber package you can access the *YouTube* API via R. The author of the package has written a short vignette to introduce some of its main functions.

Note: You can also access/open this vignette in R/*RStudio* with the following command:

```
vignette("tuber-ex")
```

Essentially, there are three main units of analysis for which you can access data with the `tuber` package:

1. Videos
2. Channels
3. Playlists

In this workshop, we will focus on accessing and working with data for individual **videos**.

Setup

```
remotes::install_github("soodoku/tuber", build_vignettes = TRUE)
# this installs the development version of tuber from GitHub
install.packages("tidyverse")
library(tuber)
library(tidyverse)
```

NB: If your version of `R` is < 4.0.0, you should also run the following command before collecting data from the *YouTube* API:

```
options(stringsAsFactors = FALSE)
```

Excursus: Keep Your Secrets Secret

If you save your API credentials in an `R` script, anyone who has that script can potentially use your API key. This can affect your quota and might even cause costs for you (if you have billing enabled for your project).

Sidenote: If you use environment variables in `R` and store your app ID and secret there, your credentials won't be shared if you share your scripts. However, the information is still stored locally in a plain text file. You can run `usethis::edit_r_environ()` to see its contents.

Excursus: Keep Your Secrets Secret

Excursus: Keep Your Secrets Secret

One way of keeping your credentials safe(r) is using the `keyring` package.

```r
install.packages("keyring")
library(keyring)
```

Excursus: The `keyring` Package

To store your credentials in an encrypted password-protected keyring you first need to create a new keyring.

```
keyring_create("YouTube_API")
```

You will then be prompted to set a password for this keyring. As always: This password should be easy to remember (for you) but hard to guess (for anyone else). You might want to write it down or store it in your password manager. In the next step, you can store your *YouTube* app secret in the keyring (*Note*: If you want to, you can also store your app ID the same way).

```
key_set("app_secret",
        keyring ="YouTube_API")
```

When you are prompted to enter the password this time you should enter your app secret. After that - as well as whenever you unlock a keyring to use the credentials stored inside - you should lock the keyring again.

```
keyring_lock("YouTube_API")
```

Authenticate Your App

```
yt_oauth(app_id = "my_app_id", # paste your app ID here
         app_secret = "my_app_secret", # paste your app secret here
         token="") # this indicates that thete is no .httr-oauth yet
```

or, if you stored your app secret in a keyring

```
yt_oauth(app_id = "my_app_id",  # paste your app ID here
         app_secret = key_get("app_secret", keyring ="YouTube_API"),
         token="")
```

Note: If you use a keyring (and have not unlocked it before), you will be prompted to enter the password for that keyring in R/*RStudio*.

Authentication

When running the `yt_oauth()` function, there will be a prompt in the console asking you whether you want to save the access token in a file. If you select "Yes", `tuber` will create a local file (in your working directory) called `.httr-oauth` to cache OAuth access credentials between `R` sessions. The token stored in this file will be valid for one hour (and can be automatically refreshed). Doing this makes sense if you know you will be running multiple `R` sessions that need your credentials (example: knitting an `R Markdown` document). Otherwise, you probably want to choose "No" here.

Note: If you use `git` in combination with *GitHub* or *GitLab* for your *YouTube* data project, you might also want to add the `.httr-oauth` file (if you create one) to your `.gitignore`.

Authentication

Similar to the app key and secret, you should never share the `.httr-oauth` as the file contains a list with your app ID and the access token. If you have created a `.httr-oauth` file and want to check its contents, you can do so as follows in R:

```r
# read the .httr-oauth file
token <- readRDS("~/.httr-oauth") # this assumes the file is stored
# app ID
token$`0123456789abcdefghij`$app$key # the token object is a list of
# app secret
token$`0123456789abcdefghij`$app$secret
# If you are unsure which token is the correct one, you can check ou
token$`0123456789abcdefghij`$app$appname
```

App Authorization (1)

After making your choice regarding the `.httr-oauth` file, a browser window should open, prompting you to select/log in with your *Google* account and to give permissions.
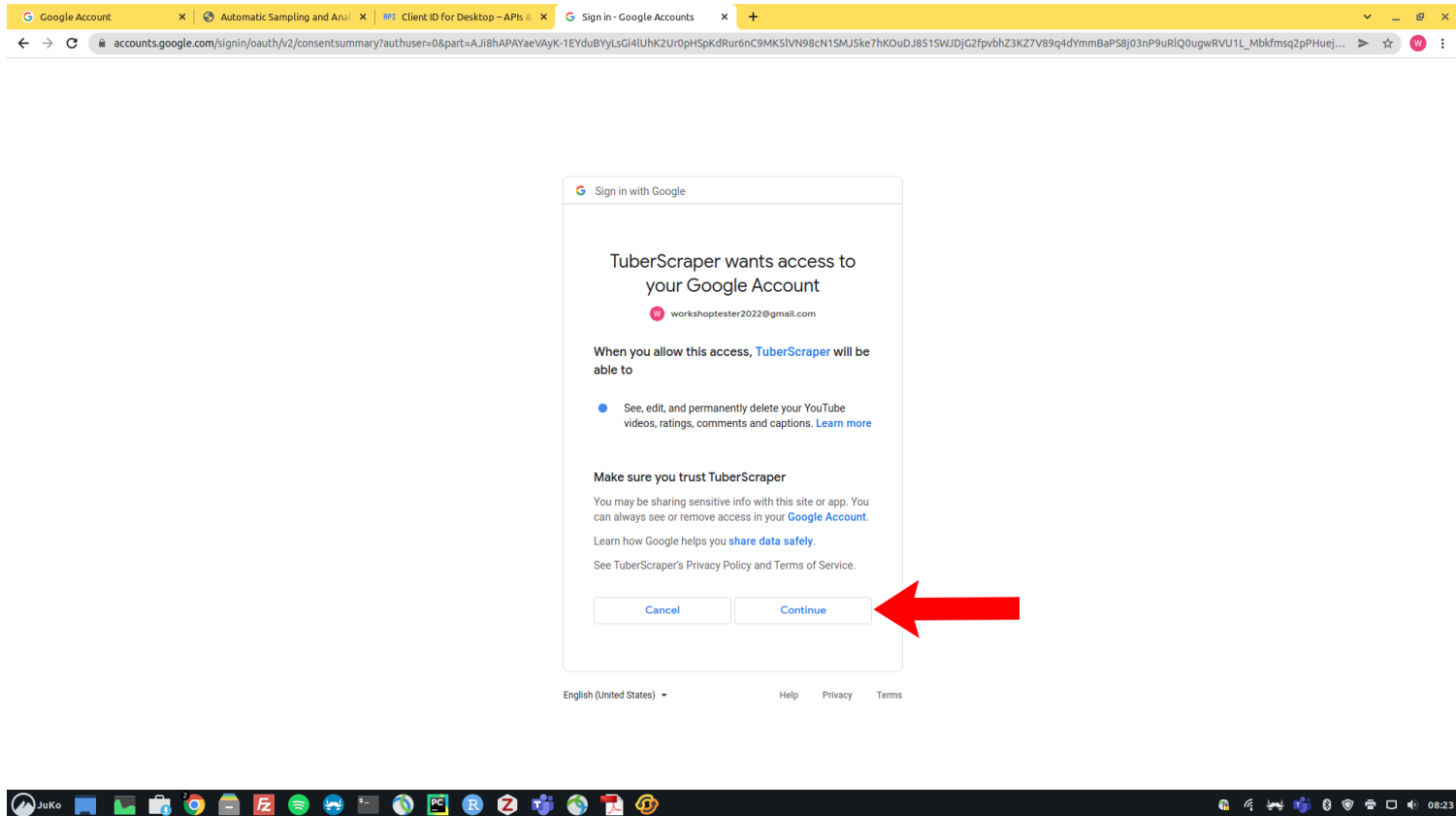
App Authorization (2)

Google warns you that the app has not been verified, but you can trust
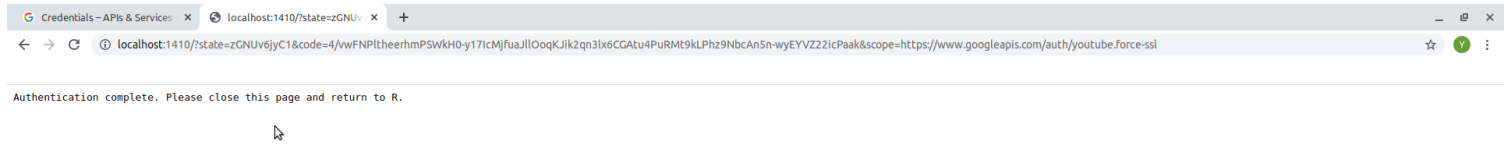yourself and click on "Continue".
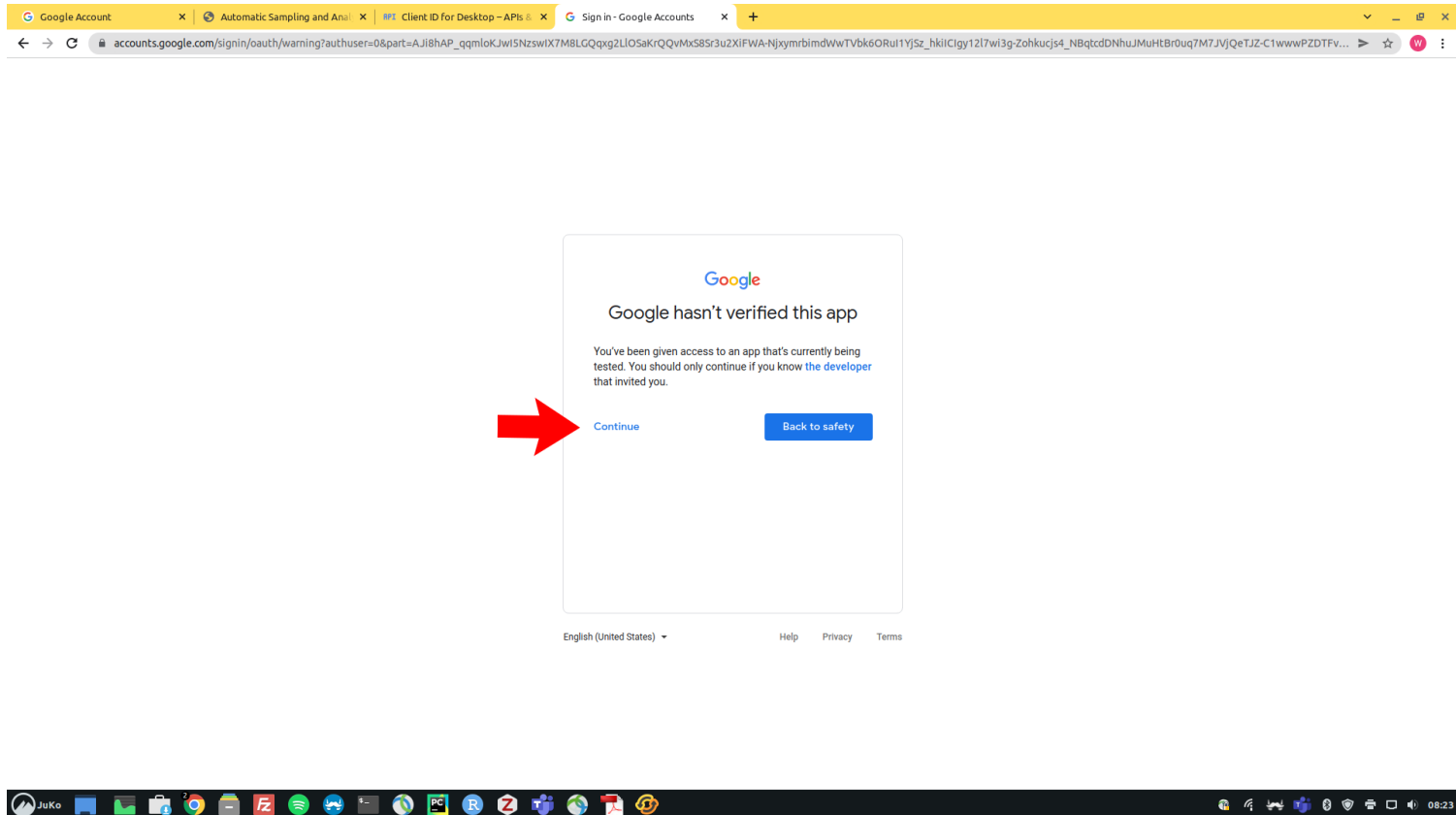
App Authorization (3)

Allow the app (and, hence, yourself) to make changes to the account.

App Authorization (4)

After that, you can close the browser tab/window and return to R/*RStudio*.

Channel Statistics (1)

After authorizing your app, you can test if everything works by getting some statistics for a channel. You can find the channel ID by clicking on the user name for a video on the *YouTube* website (the channel ID is at the end of the URL, after "/channel/"), by using the *Commentpicker* tool or directly through the API (see this Stackoverflow discussion).

```
gesis_stats <- get_channel_stats("UCiQ98odXlAkX63EaFWNjH0g")
```

```
## Channel Title: GESIS - Leibniz-Institut für Sozialwissenschaften
## No. of Views: 17293
## No. of Subscribers: 447
## No. of Videos: 71
```

Note: If you want additional data on channels/video creators and their performance on *YouTube*, you can also check out the statistics provided by *Socialblade*.

Channel Statistics (2)

`get_channel_stats()` returns a list which can then be wrangled into a dataframe (or a tibble as in the code below).

```
gesis_stats_df <- gesis_stats$statistics %>%
  as_tibble() %>%
  mutate(
    across(where(is.character), as.numeric),
    channel_id = gesis_stats$id,
    channel_name = gesis_stats$snippet$title,
    channel_published = gesis_stats$snippet$publishedAt)

gesis_stats_df
```

```
## # A tibble: 1 x 7
##    viewCount subscriberCount hiddenSubscriber~ videoCount channel_id channel_name
##       <dbl>           <dbl> <lgl>                  <dbl> <chr>      <chr>
## 1     17293             447 FALSE                     71 UCiQ98odX~ GESIS – Lei~
## # ... with 1 more variable: channel_published <chr>
```

Note: You may also want to transform the `channel_published` column to a date variable. You can, e.g., do that using the `lubridate` or the `anytime` package.

Video Statistics

To get statistics for a video you need its ID. The video IDs are the characters after the "v=" URL parameter. For example, for the video https://www.youtube.com/watch?v=DcJFdCmN98s the ID is "DcJFdCmN98s".

```r
dayum_stats <- get_stats("DcJFdCmN98s")

dayum_stats_df <- dayum_stats %>%
  as_tibble() %>%
  mutate(across(c(2:5), as.numeric))

dayum_stats_df
```

```
## # A tibble: 1 x 5
##   id           viewCount likeCount favoriteCount commentCount
##   <chr>            <dbl>     <dbl>         <dbl>        <dbl>
## 1 DcJFdCmN98s   43472949    635721             0        52618
```

Note: Until July 2021 it was also possible to get the dislike counts for videos. However, access to that via the *YouTube* API has been disabled since then. Those are also not visible anymore on the public video pages, only for the content creators.

Video Details

Using its ID, you can also get other information about a particular video, such as its description, using the `tuber` function `get_video_details()`.

```r
dayum_details <- get_video_details("DcJFdCmN98s",
                                   as.data.frame = TRUE)
# NB: The as.data.frame argument is only available in the development version of tuber

names(dayum_details)
```

```
##  [1] "response_kind"             "response_etag"
##  [3] "items_kind"                "items_etag"
##  [5] "id"                        "publishedAt"
##  [7] "channelId"                 "title"
##  [9] "description"               "thumbnails_default_url"
## [11] "thumbnails_default_width"  "thumbnails_default_height"
## [13] "thumbnails_medium_url"     "thumbnails_medium_width"
## [15] "thumbnails_medium_height"  "thumbnails_high_url"
## [17] "thumbnails_high_width"     "thumbnails_high_height"
## [19] "thumbnails_standard_url"   "thumbnails_standard_width"
## [21] "thumbnails_standard_height" "thumbnails_maxres_url"
## [23] "thumbnails_maxres_width"   "thumbnails_maxres_height"
## [25] "channelTitle"              "tags"
## [27] "categoryId"                "liveBroadcastContent"
## [29] "defaultLanguage"           "localized_title"
## [31] "localized_description"     "defaultAudioLanguage"
## [33] "pageInfo_totalResults"     "pageInfo_resultsPerPage"
```

Video Details & Statistics

Using `get_video_details()` and specifying one additional argument, you can also get a dataframe including the video details as well as the video stats.

```r
dayum_full_info <- get_video_details("DcJFdCmN98s",
                                     part = c("snippet", "statistics"),
                                     as.data.frame = TRUE)

names(dayum_full_info)
```

```
##  [1] "response_kind"              "response_etag"
##  [3] "items_kind"                 "items_etag"
##  [5] "id"                         "publishedAt"
##  [7] "channelId"                  "title"
##  [9] "description"                "thumbnails_default_url"
## [11] "thumbnails_default_width"   "thumbnails_default_height"
## [13] "thumbnails_medium_url"      "thumbnails_medium_width"
## [15] "thumbnails_medium_height"   "thumbnails_high_url"
## [17] "thumbnails_high_width"      "thumbnails_high_height"
## [19] "thumbnails_standard_url"    "thumbnails_standard_width"
## [21] "thumbnails_standard_height" "thumbnails_maxres_url"
## [23] "thumbnails_maxres_width"    "thumbnails_maxres_height"
## [25] "channelTitle"               "tags"
## [27] "categoryId"                 "liveBroadcastContent"
## [29] "defaultLanguage"            "localized_title"
## [31] "localized_description"      "defaultAudioLanguage"
## [33] "statistics_viewCount"       "statistics_likeCount"
## [35] "statistics_favoriteCount"   "statistics_commentCount"
## [37] "pageInfo_totalResults"      "pageInfo_resultsPerPage"
```

Viewer Comments

There are two `tuber` functions for collecting viewer comments for specific videos (both return a dataframe by default):

1. `get_comment_threads()` collects comments for a video. By default, it collects the 100 most recent comments. The number of comments can be changed with the argument `max_results` (needs to be >= 20). If the number is > 100, the function fetches all results (see ? `get_comment_threads`). **NB:** This function does not collect replies to comments. Also, when we tested it, the resulting dataframe only contained repetitions of the 100 most recent comments when `max_results` > 100.

2. `get_all_comments()` collects all comments for a video, including replies to comments. **NB:** This function only collects up to 5 replies per comment (the related issue in the `tuber` GitHub repo is still open). Depending on the number of comments for a video, running this function can take some time and might also deplete your API quota limit (more on that later).

Viewer Comments

Our recommendation is that if you want more than the most recent 100 comments, you should use `get_all_comments()`, even if you do not want to analyze replies to comments (you can simply filter those out later on).

If you also care about replies to comments, a good alternative to `tuber::get_all_comments()` is the `Collect()` function from the `vosonSML` package (which we will talk about in a bit) as that collects up to 20 replies for each comment (instead of just 5 as the corresponding `tuber` function does).

Exercise time 🏋️ 💪 🏃 🚴

Solutions

Collecting Comments for Multiple Videos

If you want to collect comments for multiple videos, you need a vector with all video IDs (in the code example below, this vector is named `video_ids`). If you have that, you can use `map_df()` from the purrr package to iterate over video IDs and create a combined dataframe (of course, you could also do this with a for-loop).

```
comments <- purrr::map_df(.x = video_ids,
                          .f = get_all_comments)
```

Important notice: Be aware that your daily API quota limit very likely won't allow you to collect all comments for videos with a high number of comments (more on the API quota limit in a bit).

Searching Video IDs with `tuber`

`tuber` provides the function `yt_search()` for using the API to search *YouTube*. However, using this function is extremely costly in terms of API queries/quotas.

A single search query with the `yt_search()` function can easily exceed your daily quota limit as, by default, it returns data for all search results (you can check the number of results for a search query via the *YouTube* Data API Explorer). Hence, if you want to use the `yt_search()` function, we would strongly recommend setting the `get_all` argument to `FALSE`. This returns up to 50 results.

```
yt_search(term = "tidyverse",
          get_all = F)
```

Multiple search result pages (1)

If you want to use `yt_search()` and want more than 50 search results, you can use the `page_token` argument to get more than one page of results.

```
search_p1 <- yt_search(term = "search term",
                        simplify = F,
                        get_all = F)

page_token <-search_p1$nextPageToken

search_p2 <- yt_search(term = "search term",
                        get_all = F,
                        page_token = page_token)
```

Multiple search result pages (2)

```r
# turn search_p1 from a list into a dataframe
search_p1_snippet <- lapply(search_p1$items, function(x) unlist(x$snippet))
search_p1_ids <- lapply(search_p1$items, function(x) unlist(x$id))
ids_p1_df<- purrr::map_dfr(search_p1_ids, bind_rows)
search_p1_df <- purrr::map_dfr(search_p1_snippet, bind_rows)

# combine search results from different pages into one dataframe
search_results <- bind_rows(search_p1_df, search_p2) %>%
  mutate(videoId = case_when(is.na(videoId) ~ as.character(video_id),
                             TRUE ~ as.character(videoId))) %>%
  select(-video_id) %>%
  relocate(videoId)
```

Example: Two pages of search results

```r
search_p1 <- yt_search(term = "tidyverse",
                       simplify = F,
                       get_all = F)

page_token <-search_p1$nextPageToken

search_p2 <- yt_search(term = "tidyverse",
                       get_all = F,
                       page_token = page_token)

search_p1_snippet <- lapply(search_p1$items, function(x) unlist(x$snippet))
search_p1_ids <- lapply(search_p1$items, function(x) unlist(x$id))
ids_p1_df<- purrr::map_dfr(search_p1_ids, bind_rows) %>%
  select(videoId)
search_p1_df <- purrr::map_dfr(search_p1_snippet, bind_rows) %>%
  bind_cols(ids_p1_df)
search_results <- bind_rows(search_p1_df, search_p2) %>%
  mutate(videoId = case_when(is.na(videoId) ~ as.character(video_id),
                             TRUE ~ as.character(videoId))) %>%
  select(-video_id) %>%
  relocate(videoId)

glimpse(search_results)
```

Example: Two pages of search results

```
## Rows: 100
## Columns: 17
## $ videoId                  <chr> "MKwyauo8nSI", "JtQfXY0lIzc", "1ELALQlO-yM",~
## $ publishedAt              <chr> "2019-07-24T20:44:07Z", "2019-07-01T20:12:36~
## $ channelId                <chr> "UCHSE2ppBqDNPWH0ZHeqPotw", "UCsrrSPRVwOGDt1~
## $ title                    <chr> "Intro to the Tidyverse", "Introduction to R~
## $ description              <chr> "A quick walkthrough the Tidyverse: Includes~
## $ thumbnails.default.url    <chr> "https://i.ytimg.com/vi/MKwyauo8nSI/default.~
## $ thumbnails.default.width  <chr> "120", "120", "120", "120", "120", "120", "1~
## $ thumbnails.default.height <chr> "90", "90", "90", "90", "90", "90", "90", "9~
## $ thumbnails.medium.url     <chr> "https://i.ytimg.com/vi/MKwyauo8nSI/mqdefaul~
## $ thumbnails.medium.width   <chr> "320", "320", "320", "320", "320", "320", "3~
## $ thumbnails.medium.height  <chr> "180", "180", "180", "180", "180", "180", "1~
## $ thumbnails.high.url       <chr> "https://i.ytimg.com/vi/MKwyauo8nSI/hqdefaul~
## $ thumbnails.high.width     <chr> "480", "480", "480", "480", "480", "480", "4~
## $ thumbnails.high.height    <chr> "360", "360", "360", "360", "360", "360", "3~
## $ channelTitle              <chr> "Thomas Mock", "Mark Gingrass", "RStudio", "~
## $ liveBroadcastContent      <chr> "none", "none", "none", "none", "none", "non~
## $ publishTime               <chr> "2019-07-24T20:44:07Z", "2019-07-01T20:12:36~
```

Searching Video IDs: Alternatives

There are two alternatives to using the `yt_search()` function from the `tuber` package to search for *YouTube* video IDs:

1. Manual search via the *YouTube* website: This works well for a small number of videos but is not really feasible if you want a large number of video IDs for your analyses.

2. Web scraping: Freelon (2018) argues that researchers interested in social media and other internet data should know/learn how to scrape the web in what he calls the "post-API age" (and a paper by Mancosu & Vegetti, 2020 makes a similar point). However, you should use this method with caution. The robots.txt of *YouTube* disallows the scraping of results pages. In practical terms, the (over)use of web scraping might, e.g., get your IP address blocked (at least temporarily). A complete introduction to web scraping would be beyond the scope of this workshop, but we have created some slides with a short tutorial (in the folder with the slides on the *YouTube* API) and there are many online tutorials on this (e.g., this one).

Other Useful `tuber` Functions

Function	Output
get_all_channel_video_stats('channel_id')	list with statistics for all videos in a channel
get_playlists(filter=c(channel_id='channel_id'))	list with playlists for a channel
get_playlist_items(filter=c(playlist_id='playlist_id'))	dataframe with items from a playlist

For the full list of functions in the package, see the tuber reference manual.

As always when using R (packages), it helps to check the documentation for the functions that you (want to) use. For example:

```
?get_video_details
?get_all_comments
```

YouTube API Quota Limits

In order to be able to estimate the quota costs of a specific API call via the `tuber` package, you need to know the specific queries that the function you use produces. If you want to do this, you can print the function code (you can do this in `R` by executing the function name without `()` at the end: e.g., `get_stats`). You can then use the information from that code for the Quota Calculator or the API Explorer (by checking the respective method in the YouTube API documentation). The help file for the function of interest can also be useful here (`?get_stats`).

tuber Function Code

get_stats

```
## function (video_id = NULL, ...)
## {
##     if (!is.character(video_id))
##         stop("Must specify a video ID.")
##     querylist <- list(part = "statistics", id = video_id)
##     raw_res <- tuber_GET("videos", querylist, ...)
##     if (length(raw_res$items) == 0) {
##         warning("No statistics for this video are available.\n
##         return(list())
##     }
##     res <- raw_res$items[[1]]
##     stat_res <- res$statistics
##     c(id = res$id, stat_res)
## }
## <bytecode: 0x000000002994df58>
## <environment: namespace:tuber>
```

Exemplary Quota Costs for `tuber` Functions

Function	Costs	API resource	API method	API parts
get_channel_stats('channel_id')	1	Channels	list	statistics,snippet
get_stats('video_id')	1	Videos	list	statistics
get_comment_threads(filter = c(video_id = 'video_id'), max_results = 100)	1	CommentThreads	list	snippet

The costs increase with the number of returned items/results.

For reference: `yt_search()` has a quota cost of 100 per page of results (with a default number of max. 50 results per page).

How do I Know the Quota Costs When I use `tuber`?

a) Estimate the costs based on costs for the specific query and the number of returned results. For example, if a video has 4000 comments and you wish to collect all of them, you can estimate your cost as follows: 100 results per page = 40 pages & cost per page for `get_all_comments()` = 1 -> overall cost = 40. In practice, this is a bit more difficult to estimate as both the website of the video and `get_stats()` give you the comment count including all replies, but `get_all_comments()` only returns up to 5 replies per comment.

b) If you want to check your API quota use, you can use the *Google* Developer Console: Choose your project, then select *IAM & Admin -> Quotas* from the menu on the left-hand side. The important information in the table is the one for the *YouTube Data API v3* quota for *Queries per day* (the limit should be 10000). If you want a more detailed visualizatioon, you can get that via the *Dashboard* in the *API & Services* menu: Click on the name of the API in the list on the bottom (*YouTube Data API v3*), select *Quotas* from the menu on the left, and choose *Queries per minute* from the dropdown-menu for *Queries* as well as an appropriate timeframe (e.g., 6 hrs). **NB**: It takes a bit for the numbers to update after sending your API calls.

Managing Your API Quota

The daily quota for your app resets at midnight Pacific Time. There are 3 ways of dealing with the quota limit:

1. Schedule your scripts. You can either do this manually (by splitting your data collection into batches and running the scripts on different days) or automatically (e.g., using the taskScheduleR package for Windows or the cronR package for Linux/Unix).

2. Request an increase through the Quotas menu in the developer console for your app. However, this will quite likely cost money or might not work at all.

3. Use multiple apps and/or accounts for your data collection. This requires some manual scheduling/planning. Keep in mind that there is a limited number of projects per account for the *Google* Cloud Platform. Note that - depending on your setup and use - this strategy might get your API quota, app(s), account(s) or IP address(es) suspended/blocked (at least temporarily) if employed excessively (and we can't put a number on what 'excessively' means here, so this is just a word of caution).

Alternatives to `tuber`

If you want to collect user comments, an interesting alternative to `tuber` are the packages `vosonSML` and `VOSONDash`. The focus of both of those packages is on network analysis, but they can also be used to just collect user comments via the *YouTube* API. `VOSONDash` is essentially a graphical user interface (GUI) for `vosonSML` (note for the `R` nerds: it is an `R Shiny` web application). You can install these packages as follows:

```
install.packages("vosonSML")
install.packages("VOSONDash")
```

Authentication for the *voson* packages

Unlike `tuber`, which requires the app/client ID and secret for authentication, the *voson* packages require an API key. You can create one via the *Google* Developer Console dashboard for your app.
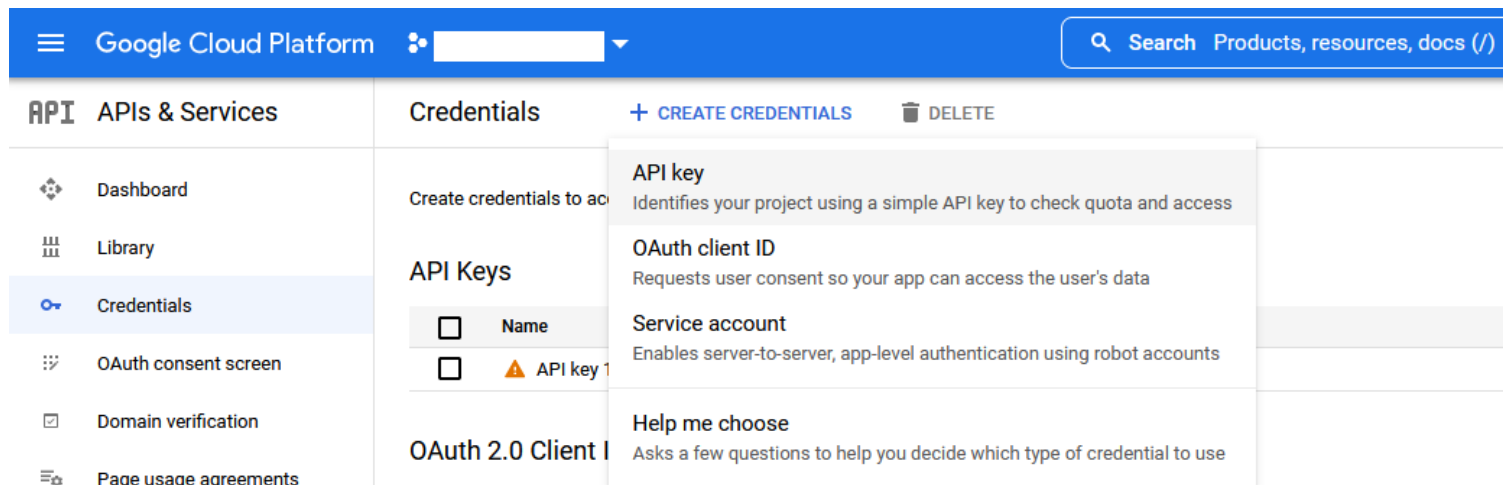
Authentication for `vosonSML`

Once you have created an API key for your app, you can authenticate as follows:

```r
library(vosonSML)

youtube_auth <- Authenticate("youtube", apiKey = "paste_your_api_key_
```

Note: The `Authenticate()` function from the `vosonSML` package only creates a credential object in your local `R` environment.

You can, of course, also use the functions from the `keyring` package for storing and accessing your API key.

```r
library(vosonSML)
library(keyring)

youtube_auth <- Authenticate("youtube",
                             apiKey = key_get("api_key",
                                              keyring = "YouTube_API"
```

Collecting comments with `vosonSML`

With the `vosonSML` function `Collect()` you can collect comments for multiple videos.

```
video_ids = c("-5wpm-gesOY", "8ZtInClXe1Q")

comments <- youtube_auth %>%
  Collect(videoIDs = video_ids,
          maxComments = 100,
          verbose = FALSE)
```

Note: The `vosonSML::Collect()` function also gives you an estimated API cost.

Collecting comments with `vosonSML`

```
## Collecting comment threads for youtube videos...
## Video 1 of 2
## ----------------------------------------------------------------
## ** video Id: -5wpm-gesOY
## -- API returned more than max comments. Results truncated to first 100 threads.
## ** Collected threads: 100
##
## Video 2 of 2
## ----------------------------------------------------------------
## ** video Id: 8ZtInClXe1Q
## -- API returned more than max comments. Results truncated to first 100 threads.
## ** Collected threads: 100
##
## ** Total comments collected for all videos 200.
## (Estimated API unit cost: 12)
## Done.
```

Collecting comments with `vosonSML`

```
glimpse(comments)
```

```
## Rows: 200
## Columns: 12
## $ Comment               <chr> "this is still the best video on youtube", "i to~
## $ AuthorDisplayName     <chr> "doomoftheend", "eggseventy", "Somnath Roy", "0x~
## $ AuthorProfileImageUrl <chr> "https://yt3.ggpht.com/ytc/AKedOLTF_qufU1laYqnXw~
## $ AuthorChannelUrl      <chr> "http://www.youtube.com/channel/UCpAZJFSj1rGvV1F~
## $ AuthorChannelID       <chr> "UCpAZJFSj1rGvV1Fo9kNveng", "UCEcTr40Lpg6V-7I9Mt~
## $ ReplyCount            <chr> "0", "0", "0", "0", "0", "0", "0", "0", "0", "0"~
## $ LikeCount             <chr> "0", "0", "0", "0", "1", "1", "0", "0", "1", "0"~
## $ PublishedAt           <chr> "2022-01-20T16:02:13Z", "2022-01-20T06:25:06Z", ~
## $ UpdatedAt             <chr> "2022-01-20T16:02:13Z", "2022-01-20T06:25:06Z", ~
## $ CommentID             <chr> "UgzmGKtfrtfbAUUNb4p4AaABAg", "Ugyn9DaUJtePpNgEH~
## $ ParentID              <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
## $ VideoID               <chr> "-5wpm-gesOY", "-5wpm-gesOY", "-5wpm-gesOY", "-5~
```

VOSONDash

As stated before, `VOSONDash` provided a GUI for using the functionalities of `vosonSML`. You can simply launch it locally by loading the library and running a command to launch the `R Shiny` app.

```
library(VOSONDash)

runVOSONDash()
```

Note: You may have to install some additional libraries to run `VOSONDash`.

Saving comment data

Regardless of whether you collected *YouTube* comments using `tuber` or `vosonSML`, you can, of course, always export them into your format of choice, such as `.rds` or `.csv`.

```r
saveRDS(comments, file = "./data/comments.rds")

readr::write_csv(comments, file = "./data/comments.csv")
```