Extracting data from APIs

What is an API (Application Programming Interface)?

- Definition:
  An API is like a "bridge" that allows two computer programs to talk to each other.
  → Think of it like a waiter in a restaurant:
    - You (the client) tell the waiter (API) what you want.
    - The waiter tells the kitchen (server).
    - The kitchen sends your food (data) back through the waiter (API).
    - Example:
      Google Maps provides an API so your app can ask for directions or map data.
      Your app doesn't need to know how Google Maps works inside — it just uses the API endpoints Google provides.

API Specification

- The API specification is like a user manual for developers.
  It describes:
    - What endpoints exist (URLs you can call)
    - What data you need to send (parameters, authentication)
    - What you'll get back (the response structure)

Endpoints

- Each endpoint is a specific "door" to a piece of functionality.
  Example (from a weather API):
    - /current → gives current weather
    - /forecast → gives future predictions

Web APIs

- These are APIs accessible over the internet.
    - Client: your app or script that makes a request.
    - Server: the system providing data.
    - Protocol: communication happens using HTTP.

HTTP (Hypertext Transfer Protocol)

- HTTP defines how data is sent and received between a client and a server over the web
- Client and Server Relationship
    - Client: The program that sends the request (e.g., your web browser, or a Python script).
    - Server: The machine that receives the request and sends back a response (e.g., a web API).
  - Key idea:
    - It is always the client that starts the communication.
      The server never contacts the client first.
  - Example:
    - Client → sends request: "Give me today's weather in London."
    - Server → replies: "Here's the weather data."
- HTTP is Stateless
  - Each request is independent.
  - The server doesn't remember previous requests from the same client.
  - Think of it like sending separate letters in the mail — each one is processed individually, even if they're from the same person.
- HTTP Cookies
  - Since HTTP is stateless, websites use cookies to keep context between multiple requests.
  - A cookie is a small piece of data stored in your browser (or client).
  - It's sent automatically with each subsequent request to the same server.
  - Example:
    In an online store:
    You add an item to your cart (1st request).
    You add another item (2nd request).
    Cookies help the website remember what's already in your cart — otherwise, it would "forget."

API Terminology

- HTTP Method
  - This defines what kind of action you want to perform on the API resource.
  - Two key methods:

Method	Purpose	Example
GET	Retrieve (read) data from the server	Get all planets
POST	Send (create) new data to the server	Add a new planet

  - Example analogy:
  - GET → "Tell me what's in the database."
  - POST → "Add this new item to the database."

- Base URL
  - The main entry point of the API.
    It's like the domain of the API website.
  - Example:

arduino Code kopieren

```
https://planets-by-api-ninjas.p.rapidapi.com
```

- Path
  - Specifies which resource you want to access.
    It comes after the base URL.
  - Example:

bash Code kopieren

```
/v1/planets
```

This path tells the API: "I want the list of planets."

- Endpoint
  - This is the full URL that points to a specific resource or service.
  - Endpoint = Base URL + Path

Example:

bash Code kopieren

```
https://planets-by-api-ninjas.p.rapidapi.com/v1/planets
```

You'd call this endpoint to extract information about planets.

- Query Parameters
    - Used to filter, sort, or customize your request.
      They appear after a ? in the URL as key–value pairs.
    - Example:

    - Here, name=Mars tells the API to return information only about Mars.
    - You can add multiple parameters by using &, like:
      ?name=Mars&atmosphere=thin

- Headers
    - These provide metadata about your request — extra info that isn't part of the data itself.
    - Common header uses:
        - Authentication (API keys, tokens)
        - Content type (e.g., JSON, XML)

- Body
    - The body carries more complex data — mainly used in POST requests.
    - It's not part of the URL; it's sent inside the request.
    - Usually in JSON format.

✅ **Summary Table**

Term	Description	Example
HTTP Method	Action type	GET / POST
Base URL	Main API address	`https://api.example.com`
Path	Specific resource	`/v1/data`
Endpoint	Base URL + Path	`https://api.example.com/v1/data`
Query Parameters	Filters in URL	`?name=Mars`
Headers	Extra info	Auth, Content-Type
Body	Sent data (JSON)	`{"name": "Earth"}`

HTTP Flow: How a Request Travels

1. Client opens a TCP connection
    - TCP (Transmission Control Protocol) is one of the core internet protocols — it ensures reliable delivery of data between computers.
    - Think of TCP as the "phone line" that must be connected before two people can talk.
    - Once the TCP connection is established, both sides can send data packets safely.
    - Example:
      When you visit a website, your browser first opens a TCP connection to the web server (usually on port 80 for HTTP or 443 for HTTPS).
2. Client sends an HTTP message
    - Once connected, the client sends an HTTP request.
    - The request includes:
        - HTTP method (e.g., GET, POST)
        - Path and query parameters
        - Headers
        - (Optional) body data
3. Server sends a response
    - The server reads the request, processes it, and sends back an HTTP response.
    - The response contains:
        - A status code (e.g., 200 OK, 404 Not Found, 500 Internal Server Error)
        - Headers (e.g., content type)
        - Body (actual data, usually in JSON or XML)
4. Client closes or reuses the connection
    - After getting the response, the client can:
    - Close the TCP connection, or
    - Reuse it for another request (this is called a persistent connection, and it saves time).

- Most modern web applications use persistent connections for efficiency.

GET

- Purpose
  - The GET method is used to retrieve data from a server.
  - It's a read-only operation — meaning it does not change anything on the server (no updates, no deletions, no new data created).
- No Request Body
  - A GET request does not have a body.
  - All the necessary information (like filters or search terms) is included in the URL.
- Parameters in the URL
  - You can modify the URL to request specific or filtered data.
  - These parameters appear after the question mark (?).
- Headers
  - Headers send extra info with the request, such as:
  - Desired format of the response (Accept: application/json)
  - Authentication info (API keys or tokens)
- Case Insensitive
  - HTTP headers and methods are not case sensitive.
  - So GET, get, and Get all mean the same thing.
- Unknown Headers Are Ignored
  - If the server doesn't recognize a header, it simply ignores it — it won't crash or throw an error.

Investigate a request

- If we run this in R

```
4  # GET dry run examples ------------------------------------------------------------
5  local_url <- example_url()
6  local_url
7  req <- request(local_url)
8  req %>%
9    req_dry_run()
```

  Example_url() -> We need an URL for the request. Here we use the "webfakes" R package to simulate a web server. Answer yes, to install the "webfakes" package

- Outcome

```
> # GET dry run examples ------------------------------------------------------------
> local_url <- example_url()
> local_url
[1] "http://127.0.0.1:57639/"
> req <- request(local_url)
> req %>%
+   req_dry_run()
GET / HTTP/1.1
accept: */*
accept-encoding: deflate, gzip
host: 127.0.0.1:57639
user-agent: httr2/1.2.1 r-curl/7.0.0 libcurl/8.14.1
```

  GET -> We use the GET method

  HTTP/1.1 -> Use the standard HTTP/1.1 protocol for communicatio

  Host -> Host IP and port (i.e., the address of the API

  User-agent -> The software making the request (both R and and C libraries are involved)

GET request with headers

```
# GET with parameter and header ----------
req <- request("http://165.22.92.178:8080") %>%
  req_url_path("fib") %>%
  req_url_query(n = 7) %>%
  req_headers(authorization = "DM_DV_123#!")
req %>% req_dry_run()
resp <- req %>%
  req_perform()
resp %>%
  resp_body_string()
```

This is the base URL of your API (the "entrance"). It points to your server (the droplet of the professor).
The path defines which endpoint or function of the API you're using., Here, the path "fib" means you're calling the Fibonacci endpoint.
This tells the API: "Please give me the Fibonacci number for n = 7.
This adds a header called authorization, which usually contains a token or key for authentication. This ensures only authorized users can access your API.
This shows what the full request would look like without actually sending it.
Now the request is actually sent to the server. The result (resp) contains: HTTP status, Headers, Body
The response body (the actual data) is read as a string.

"fib" explained:

- /fib → returns Fibonacci numbers
- /weather → might return weather data
- /users → might return user data

- Outcome

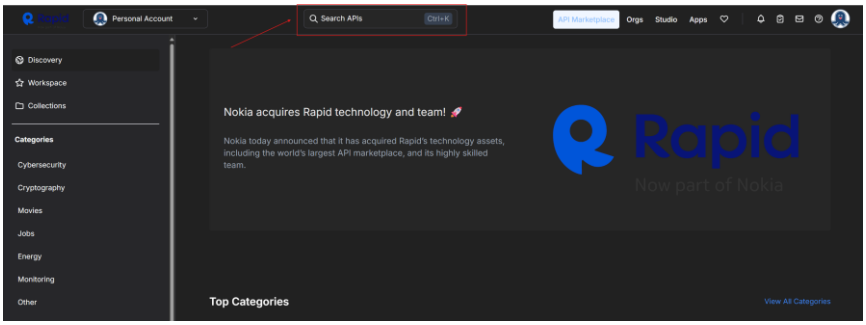
```
> # GET with parameter and header ----------
> req <- request("http://165.22.92.178:8080") %>%
+   req_url_path("fib") %>%
+   req_url_query(n = 7) %>%
+   req_headers(authorization = "DM_DV_123#!")
> req %>% req_dry_run()
GET /fib HTTP/1.1
accept: */*
accept-encoding: deflate, gzip
authorization: <REDACTED>
host: 165.22.92.178:8080
user-agent: httr2/1.2.1 r-curl/7.0.0 libcurl/8.14.1

> resp <- req %>%
+   req_perform()
> resp %>%
+   resp_body_string()
[1] "13"
```

The number 13 is not random — it's the mathematical result of your API processing the request for the 7th Fibonacci number.

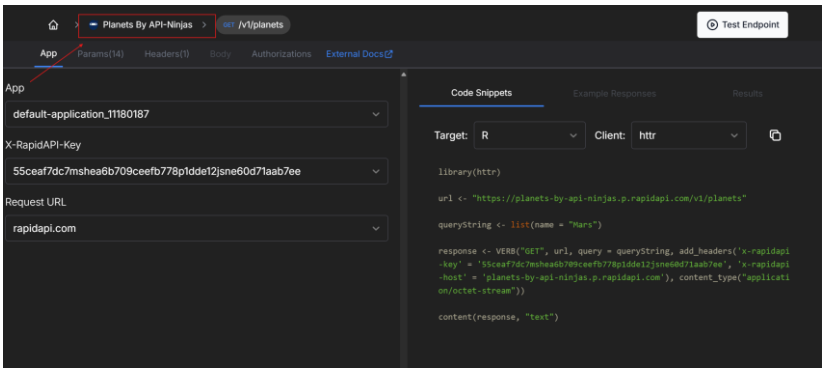
Using RapidAPI.com

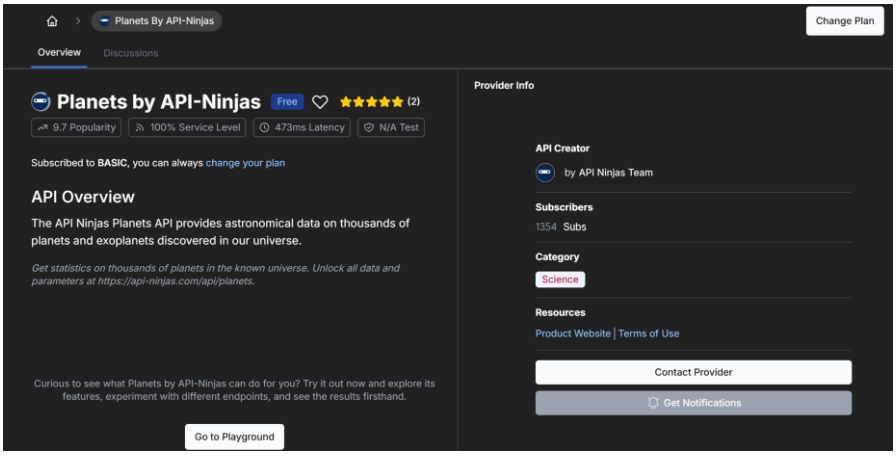
- What is RapidAPI?
    - RapidAPI is like an “app store for APIs.”
      It gathers thousands of APIs from many providers and lets you:
        - Browse them easily
        - Try them out in the browser
        - Subscribe to use them in your own code
- Think of it as a one-stop shop where you can find APIs for:
    - Weather data
    - Sports scores
    - Translation
    - Finance and crypto data
    - Space/planets info
    - Machine learning tools
- Many APIs on RapidAPI offer a free tier — meaning you can use them without paying, but with limits.
- The rate limit tells you how many requests you can make per minute, hour, or day.

Search for an API



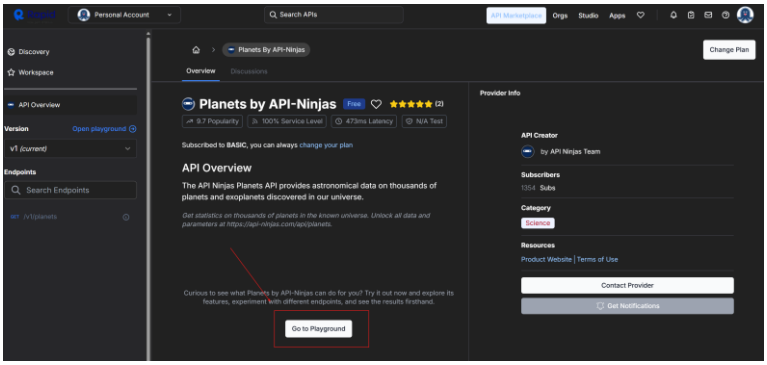
Inspect/get info about api



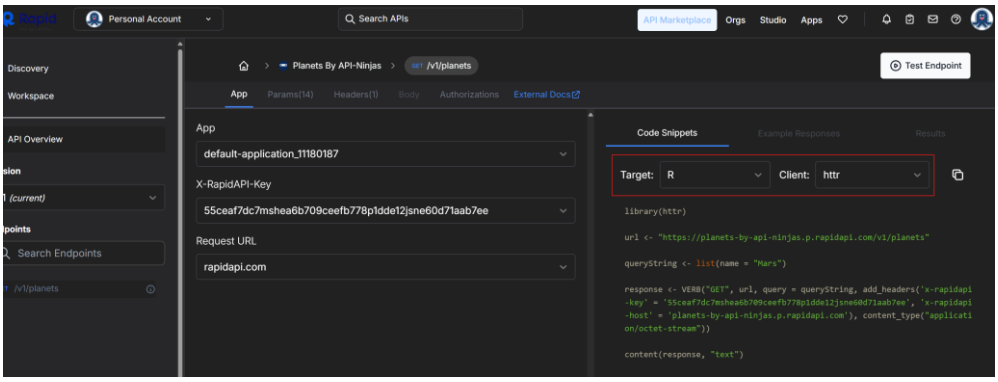


Setting up the request in R

Go to playground



Choose the target and client to get an example code in R



-> here httr package is used which is an old package we need httr2

Ask chatgpt to change from httr to httr2

Get all the information about the planet Mars

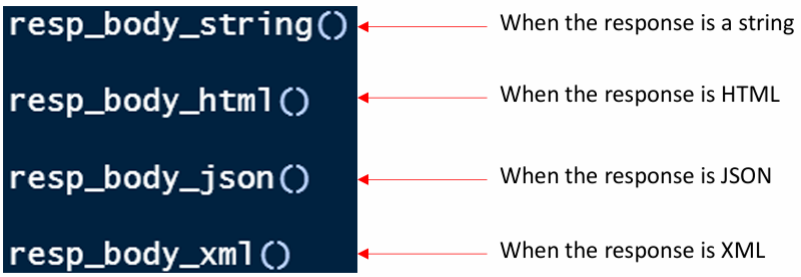
```
> library(httr2)
>
> url <- "https://planets-by-api-ninjas.p.rapidapi.com/v1/planets"
>
> resp <- request(url) |>
+   req_url_query(name = "Mars") |>
+   req_headers(
+     "x-rapidapi-key" = "55ceaf7dc7mshea6b709ceefb778p1dde12jsne60d71aab7ee",
+     "x-rapidapi-host" = "planets-by-api-ninjas.p.rapidapi.com",
+     "Content-Type" = "application/octet-stream"
+   ) |>
+   req_perform()
>
> resp_body_string(resp)
[1] "[{\"name\": \"Mars\", \"mass\": 0.000338, \"radius\": 0.0488, \"period\": 687.0, \"semi_major_axis
\": 1.542, \"temperature\": 210.0, \"distance_light_year\": 3.7e-05, \"host_star_mass\": 1.0, \"host_st
ar_temperature\": 6000.0}]"
```

If you change the name of the planet you get information about another planet

```
> url <- "https://planets-by-api-ninjas.p.rapidapi.com/v1/planets"
>
> resp <- request(url) |>
+   req_url_query(name = "Venus") |>
+   req_headers(
+     "x-rapidapi-key" = "55ceaf7dc7mshea6b709ceefb778p1dde12jsne60d71aab7ee",
+     "x-rapidapi-host" = "planets-by-api-ninjas.p.rapidapi.com",
+     "Content-Type" = "application/octet-stream"
+   ) |>
+   req_perform()
>
> resp_body_string(resp)
[1] "[{\"name\": \"Venus\", \"mass\": 0.00257, \"radius\": 0.0847, \"period\": 224.7, \"semi_major_axis
\": 0.723332, \"temperature\": 737.0, \"distance_light_year\": 4e-06, \"host_star_mass\": 1.0, \"host_s
tar_temperature\": 6000.0}]"
```

-> Here Venus

Resp_body_String(resp) -> We can interpret the body from different types of format



POST

- Purpose
  - The POST method is used to send data to a server — usually to create, update, or process something.
  - It's like submitting a form on a website or uploading data to be processed.
- Request Structure
  - A POST request contains:
  - Headers → Metadata about the request (like authentication or content type)
  - Parameters (optional) → Small pieces of info in the URL
  - Body → The main content being sent — also called the payload
- The Body (Payload)
  - The body is where you include the data you want to send to the server.
  - This data is often in JSON format, but can also be text, binary, or files.
  - In R, the payload could even come from a data frame or any R object converted to JSON.
- What Happens on the Server
  - The server receives the payload (the body data).
  - It can then:
    - Store the data (e.g., create a new user account)
    - Process it (e.g., run a machine learning model on your dataset)
    - Return a response (e.g., "User created successfully" or model results)
  - Example:
    Imagine your API wraps an R function that receives data, runs a regression, and returns the predicted values — that's a POST request in action.
- Can Modify or Process Data
  - Unlike GET (which only reads data), POST can change or create data on the server — though not always.
    Examples:

📄 **Examples:**

Action	Example
Create a blog post	`/posts` → new post added
Create a user	`/users` → new account created
Submit data for analysis	`/predict` → returns model output

post run

```r
# POST dry run example ------------------------------------------------------------
# Body as an R list
req_body <- list(x = c(1, 2, 3), y = c("a", "b", "c"))
req_body
# Body transformed to JSON in the request
request(example_url()) %>%
  req_body_json(req_body) %>%
  req_dry_run()
```

Create a body (the data you want to send)
This creates an R list — your payload.
x contains numeric values [1, 2, 3]
y contains character values ["a", "b", "c"

request(example_url()) creates a new request to some test API (the function example_url() just provides a demo URL).

req_body_json(req_body) converts your R list into JSON format and attaches it as the body of the request.

This command does not send the request. Instead, it prints out what your request would look like if you did send it.

Outcome

```
> # POST dry run example ------------------------------------------------------------
> # Body as an R list
> req_body <- list(x = c(1, 2, 3), y = c("a", "b", "c"))
> req_body
$x
[1] 1 2 3

$y
[1] "a" "b" "c"

> # Body transformed to JSON in the request
> request(example_url()) %>%
+   req_body_json(req_body) %>%
+   req_dry_run()
POST / HTTP/1.1
accept: */*
accept-encoding: deflate, gzip
content-length: 31
content-type: application/json
host: 127.0.0.1:57639
user-agent: httr2/1.2.1 r-curl/7.0.0 libcurl/8.14.1

{
  "x": [
    1,
    2,
    3
  ],
  "y": [
    "a",
    "b",
    "c"
  ]
}
>
```

Data we created above

You're making a POST request

You're sending JSON data

The target server

Example of processing complex data

```r
# POST with data ------------------------------------------------------------
# Simulate data
n <- 100
x1 <- rnorm(n = n)
x2 <- rnorm(n = n)
x3 <- rnorm(n = n)
y <- 2*x1 + 3*x2 + 2*x3 + rnorm(n = n)
df <- round(data.frame(y = y, x1 = x1, x2 = x2, x3 = x3))
# Construct a request including the data
req <- request("http://165.22.92.178:8080") %>%
  req_url_path("lm") %>%
  req_body_json(as.list(df)) %>%
  req_headers(authorization = "DM_DV_123#!")
# Send the request to the API
resp <- req %>%
  req_perform()
# View the result
resp %>%
  resp_body_json()
```

You generate 3 predictors (x1, x2, x3) — each 100 random numbers from a normal distribution

You generate a dependent variable (y) that's a linear combination of them

You then combine everything into a data frame (df), and round it for simplici

The base URL (API server on a droplet)

Calls the /lm endpoint — probably an R linear model API

Converts your dataframe into a JSON body and attaches it to the request

Adds an authorization header (to prove you're allowed to access the API)

This line executes the POST request — your data is actually sent to the server.
The server's job is now to: Receive your JSON data, Convert it back into an R data frame, Run lm(y ~ x1 + x2 + x3) internally, Return the model output (probably coefficients or summary).

This line reads the response body and automatically converts it from JSON back to an R object (list or data frame).

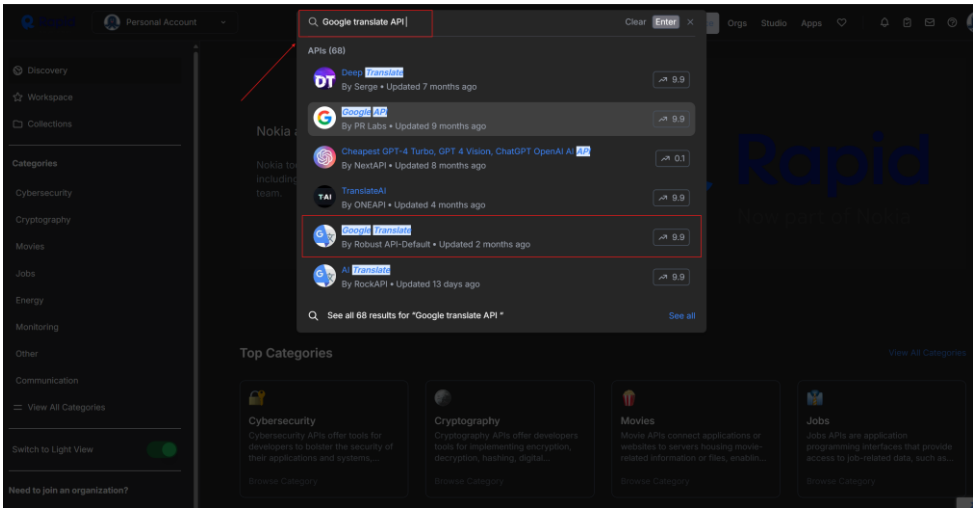
Outcome

```
> # POST with data ------------------------------------------------------------
> # Simulate data
> n <- 100
> x1 <- rnorm(n = n)
> x2 <- rnorm(n = n)
> x3 <- rnorm(n = n)
> y <- 2*x1 + 3*x2 + 2*x3 + rnorm(n = n)
> df <- round(data.frame(y = y, x1 = x1, x2 = x2, x3 = x3))
> # Construct a request including the data
> req <- request("http://165.22.92.178:8080") %>%
+   req_url_path("lm") %>%
+   req_body_json(as.list(df)) %>%
+   req_headers(authorization = "DM_DV_123#!")
> # Send the request to the API
> resp <- req %>%
+   req_perform()
> # View the result
> resp %>%
+   resp_body_json()
$`(Intercept)`
[1] -0.026

$x1
[1] 2.0457

$x2
[1] 2.7078

$x3
[1] 1.8324
```

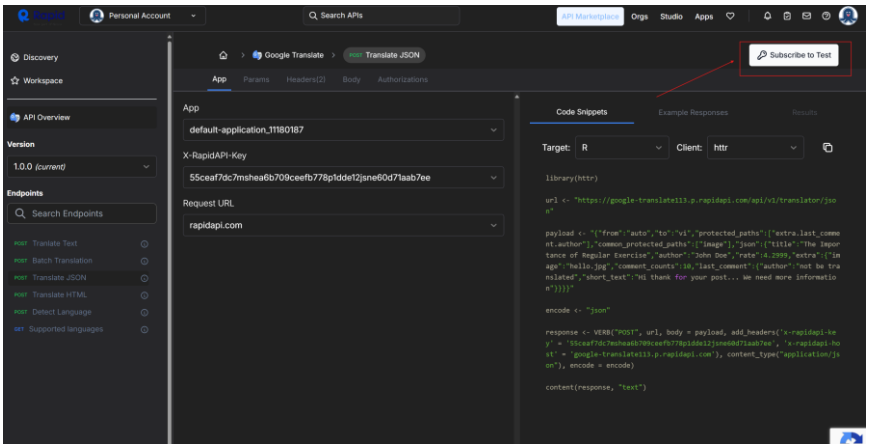
You used an API as a remote calculator.
Instead of doing the math on your own computer, you sent your data to another computer (the API server).
That computer ran the calculation (the regression) and sent the results back to you.

POST with rapid API (google Transalte)

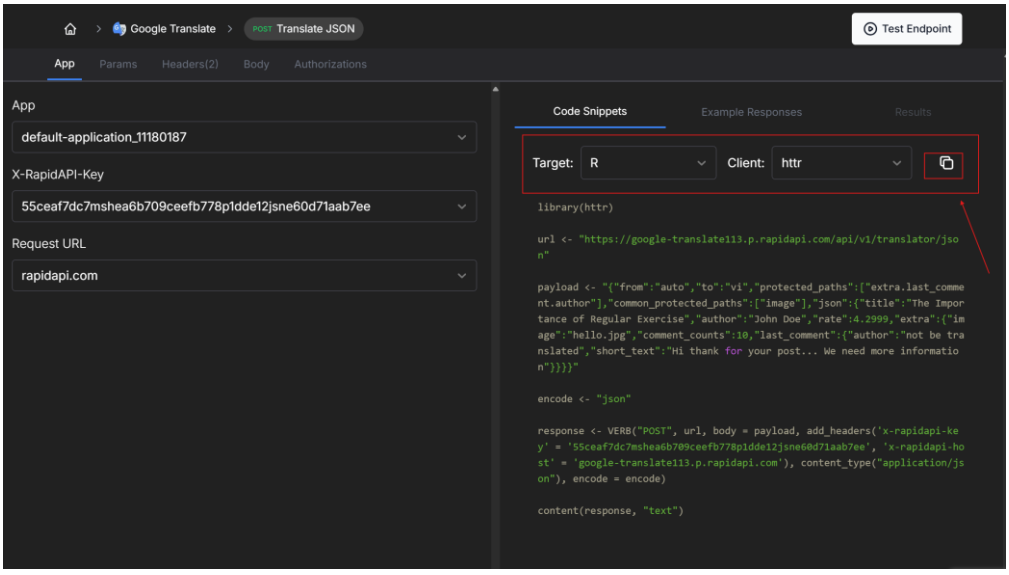
In rapid look for Google Translate



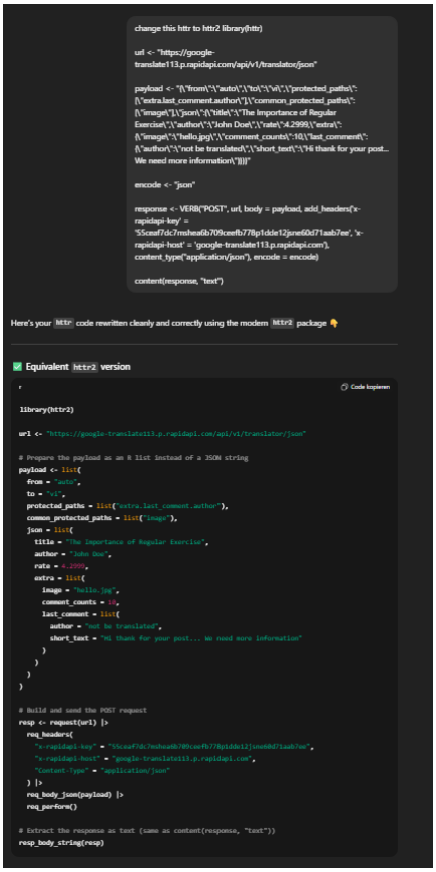
Subscribe to it



Copy the code



Ask chatgpt to change from httr to httr 2



We only want to send it from one language to another so we can remove the other parameters

✅ Final Code

r Code kopieren

```r
library(httr2)

url <- "https://google-translate113.p.rapidapi.com/api/v1/translator/text"

payload <- list(
  from = "auto",    # detect language automatically
  to = "vi",        # translate to Vietnamese
  text = "The Importance of Regular Exercise"  # your text to translate
)

resp <- request(url) |>
  req_headers(
    "x-rapidapi-key" = "55ceaf7dc7mshea6b709ceefb778p1dde12jsne60d71aab7ee",
    "x-rapidapi-host" = "google-translate113.p.rapidapi.com",
    "Content-Type" = "application/json"
  ) |>
  req_body_json(payload) |>
  req_perform()

# Show the translated text as JSON (R list)
resp_body_json(resp)
```

Copy the code and adapt in R we want to translate from englisch to Danish and another text

And add result$trans at the end

```r
library(httr2)

url <- "https://google-translate113.p.rapidapi.com/api/v1/translator/text"

payload <- list(
  from = "en",    # detect language automatically
  to = "da",       # translate to Vietnamese
  text = "Hello World"  # your text to translate
)

resp <- request(url) |>
  req_headers(
    "x-rapidapi-key" = "55ceaf7dc7mshea6b709ceefb778p1dde12jsne60d71aab7ee",
    "x-rapidapi-host" = "google-translate113.p.rapidapi.com",
    "Content-Type" = "application/json"
  ) |>
  req_body_json(payload) |>
  req_perform()

# Show the translated text as JSON (R list)
resp_body_json(resp)

result$trans
```

The API/URL from rapid Api which is google translator

Here we select what we want to achieve in this case from english to danish and the text "hello World"

we get this directly fro rapid API

My Rapid Api Key

My Rapid API host

The Type of content

Changes the list above to json

Shows the translation

Outcome

```
> library(httr2)
>
> url <- "https://google-translate113.p.rapidapi.com/api/v1/translator/text"
>
> payload <- list(
+   from = "en",    # detect language automatically
+   to = "da",       # translate to Vietnamese
+   text = "Hello World"  # your text to translate
+ )
>
> resp <- request(url) |>
+   req_headers(
+     "x-rapidapi-key" = "55ceaf7dc7mshea6b709ceefb778p1dde12jsne60d71aab7ee",
+     "x-rapidapi-host" = "google-translate113.p.rapidapi.com",
+     "Content-Type" = "application/json"
+   ) |>
+   req_body_json(payload) |>
+   req_perform()
>
> # Show the translated text as JSON (R list)
> resp_body_json(resp)
$trans
[1] "Hej Verden"

>
> result$trans
[1] "Hej verden!"
```

In rapid Api we can find current subscriptions