Extract information from JSON data

Silvie Cinková

2025-12-15

Table of contents

```
library(tidyverse)
if(system.file(package = "jsonlite") == "") {
  cat("Package 'jsonlite' is needed. Installing.\n")
}
```

```r
if (require("jsonlite") == FALSE) {
  library(jsonlite)
}
```

1 JSON

- **J**ava**S**cript **O**bject **N**otation

```
{"array": [ "object1":
              {"name1": "property-string/bool",
                "name2": number},
            "object2": {"name222": "property..."}
]
}
```

JSON is one of the standard data interchange formats, like e.g. XML or HTML. It originates from JavaScript, and it corresponds to its native data structures, but it became popular across platforms because it is human-readable and technically practical at the same time. Most programming languages have libraries to translat JSON data into its own native structures. For instance, the default interpretation of a JSON file in R is **list**.

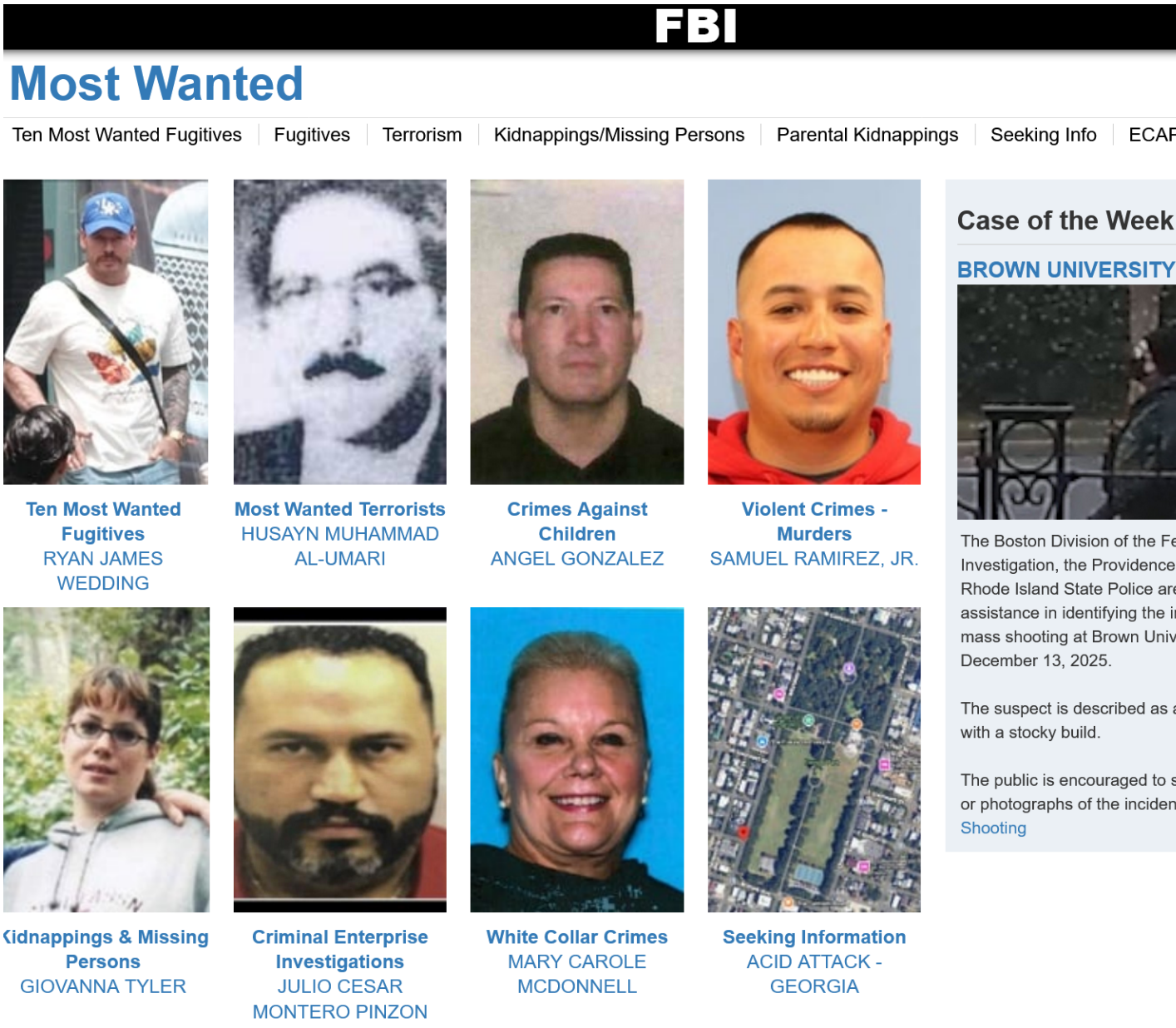
Syntax in a nutshell:

`[` contains an **array** of **objects** or other arrays. Each object is enclosed in `{` and contains *name-property* (aka *attribute-value*) pairs. Sometimes the names/attributes are also called *keys*. Letter case and punctuation matter, indentation does not.

In JSON, values must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array
- a boolean
- *null*

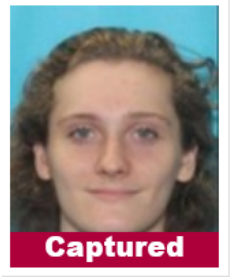
2 FBI list of Wanted Persons

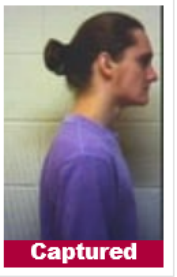


Many websites display information from an underlying database. Sometimes they provide an *API*, (Automated Programming Interface), from where you can export the database data. The typical export format is JSON, even for data that the website renders as tabular.

Our example: FBI list of wanted persons (criminals, witnesses, victims alike).

3 Individual info



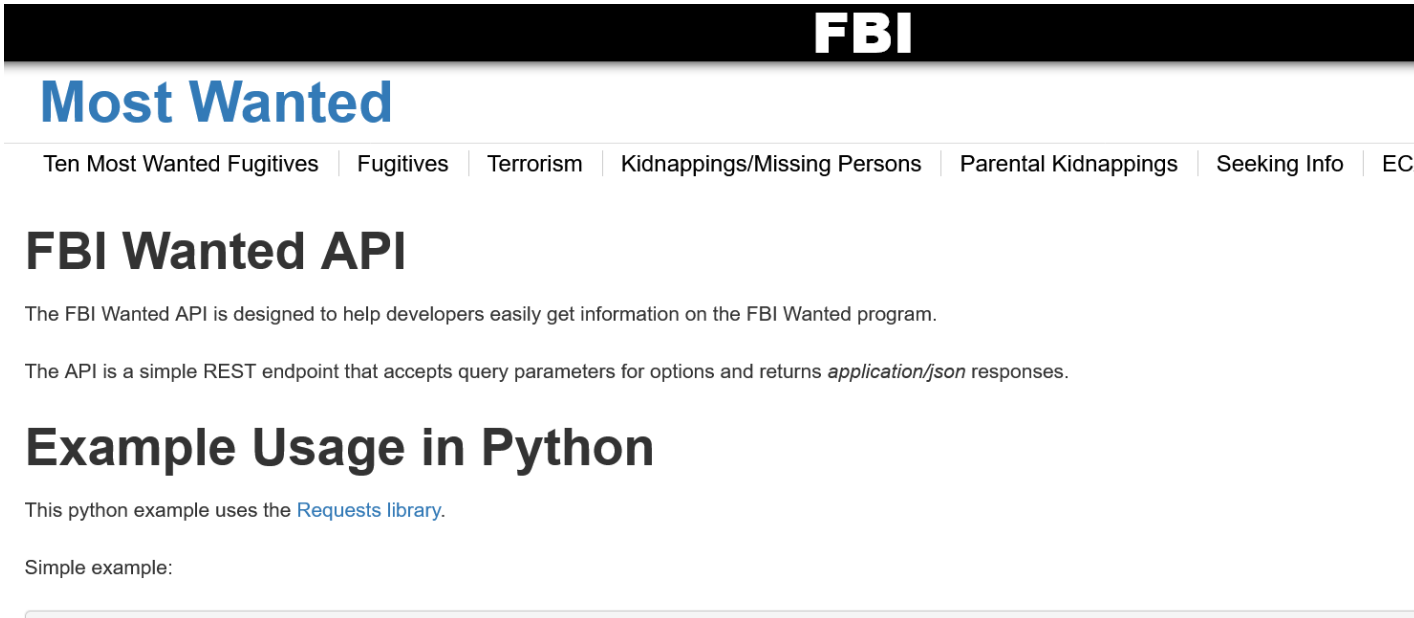


Date(s) of Birth Used	Dece
Hair	Blonc
Eyes	Blue
Height	5'8"
Weight	180 p
Sex	Male
Race	White

When you click on an item, you are going to see something like this. You anticipate that this web is connected to a real database and you wonder whether they will let you have the data directly from the database, so you could for instance filter it as you like.

4 FBI API

https://www.fbi.gov/wanted/api



FBI

Most Wanted

Ten Most Wanted Fugitives | Fugitives | Terrorism | Kidnappings/Missing Persons | Parental Kidnappings | Seeking Info | EC

FBI Wanted API

The FBI Wanted API is designed to help developers easily get information on the FBI Wanted program.

The API is a simple REST endpoint that accepts query parameters for options and returns *application/json* responses.

Example Usage in Python

This python example uses the Requests library.

Simple example:

```
import requests
import json

response = requests.get('https://api.fbi.gov/wanted/v1/list')
data = json.loads(response.content)
print(data['total'])
print(data['items'][0]['title'])
```

Example providing search parameters:

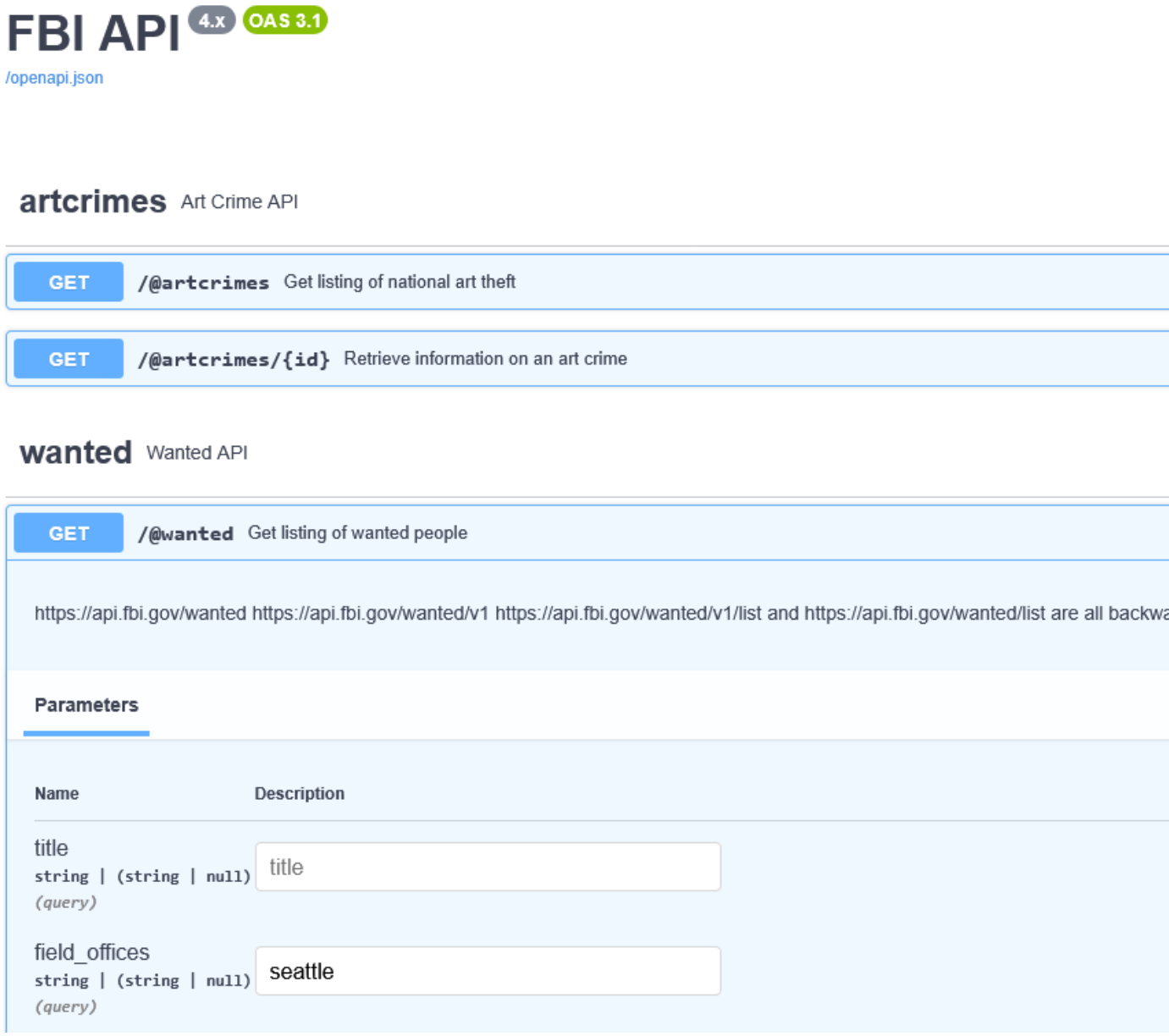
They indeed do. The FBI website has an API that allows you to download their lists of wanted persons. Many APIs have extensive documentation, where they list URLs of all possible commands they let you execute. This API only documents two parameters, and not even in a proper documentation but implicitly in examples from a Python code.

From that code (beyond this slide) you can infer that you can filter the items by "field offices" and that the output is *paginated*. That means that they will give you the data in batches fitting one scrollable html page and you will have to submit a separate API request to get each page of the records files.

If you read the entire documentation of the FBI Wanted API, you will see that the Python code puts the additional parameters into a special argument. This is also possible when you write the corresponding code in R, but you can as well add these additional parameters directly into the API URL. In the following slides, we will inspect diverse URLs directly in the web browser, without worrying about coding the requests in R.

5 Interactive demo

https://api.fbi.gov/docs#/wanted/_wanted___wanted_get



There is one better documentation here: https://api.fbi.gov/docs#/wanted/_wanted___wanted_get. There you can see more (all?) implemented parameters. The page is interactive; you can place a request by filling out a form. They will also show you a Unix shell script that will perform the request programmatically.

6 Experiment with parameters

the basic api url: https://api.fbi.gov/wanted/v1/list

Concatenate parameters:

- first param: use `?param=value`
- 2nd+ param: use `&param=value`

https : / / api.fbi.gov / wanted / v1 / list?race = black&sex = female&poster_classification = missing

Adding parameters to the base API request has its own syntax.

- The first parameter starts with a question mark. Then comes the name of the parameter (key), an equal sign, and then the value.

To see the results of the API requests in the slide, copy paste them into your web browser and hit Enter. You will see the data right away.

7 Encode URL

- Diacritics, punctuation, spaces!!!
- reserved characters `TRUE`, too

```
URLencode("Bärbel Müller?!", repeated = FALSE, reserved = TRUE)
```

```
[1] "B%C3%A4rbel%20M%C3%BCller%3F%21"
```

```
URLencode("Bärbel Müller?!", repeated = FALSE, reserved = FALSE)
```

```
[1] "B%C3%A4rbel%20M%C3%BCller?!"
```

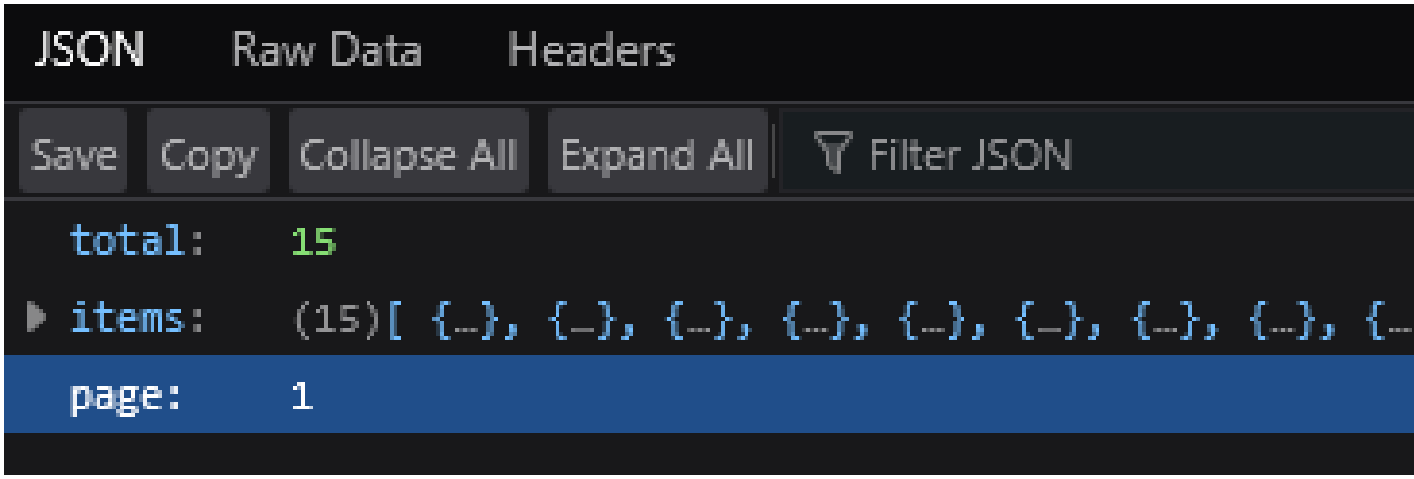
This is more a side note for the case that you need to put in some real text in the URL request. Some libraries and some browsers will even do it for you. But when not, you will be prepared.

When you want to add to the URL a parameter value that contains diacritics, spaces, or punctuation, you must *URL encode* them. You can of course do it programmatically in R. Most APIs need UTF-8 URL encoding. Outdated: Latin-1 or Windows 1252 encoding. The

R function `utils::URLencode` encodes in UTF-8, up-to-date. Double check with free online URL encoders. If you get two different outcomes, take the longer one. Why? UTF-8 encodes diacritics as a separate character. Hence for instance *ä* will be encoded as `%C3%A4` in UTF-8, which are two positions escaped with `%`. On the other hand, the same letter will be encoded with one escaped character in Latin-1, like this: `%E4`. For more info, see https://www.w3schools.com/tags//ref_urlencode.asp .

8 Structure of FBI Wanted API responses

https://api.fbi.gov/wanted/v1/list?field_offices=seattle



The slide shows a browser window with the response when you run the request above by copy-pasting it into a browser window.

It is not the raw data but the "JSON" view where the data appears somewhat prettified with html.

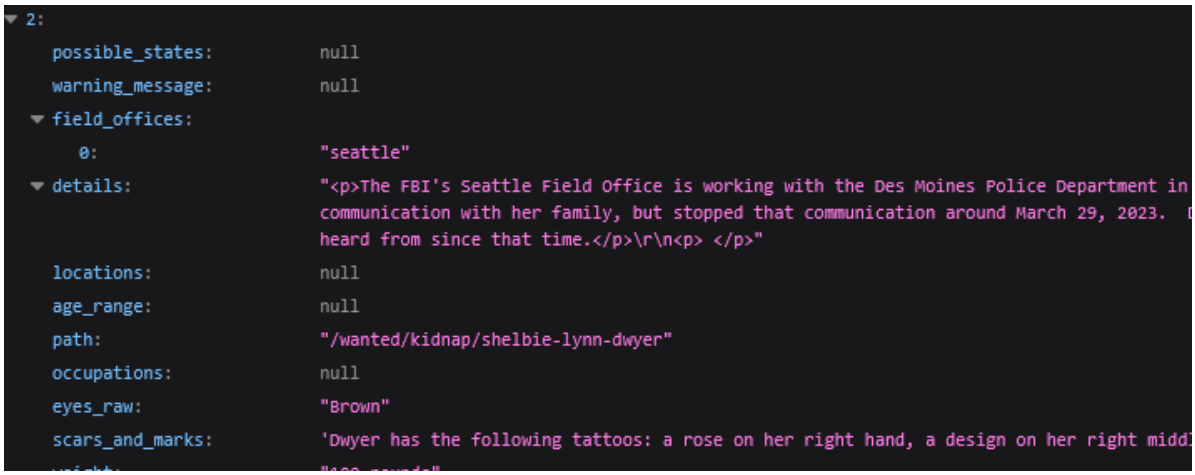
You can see and download the raw data by switching to the "Raw" tab in the browser window.

Data view with all elements (*total*, *items*, *page*) collapsed (hit the "Collapse All" button).

- `total` is a key-value pair. (*key* = `total`, *15* = `value`) . How many pages the response takes in total.
- `items` is an `[ array]` of `{objects}`. Each object represents one wanted person.
- `page` is again a key-value pair.

> **Object vs. Object literal**
>
> JSON comes from the Java Script programming language. In this context, it makes sense to make a distinction between the actual object and the way it is represented, i.e. the object literal. When in Java Script, you save a Java Script literal in a variable, you get a Java Script object. Nevertheless, outside of Java Script programming we meet only *object literals*, and this gets often abbreviated to just *object*. Hopefully it does not matter too much if we stick to this simplification.

9 Objects (FBI Wanted) in the `items` list



Let us zoom in on the `items` element. It is an array of objects. We are looking at one of the objects within the `items` array.

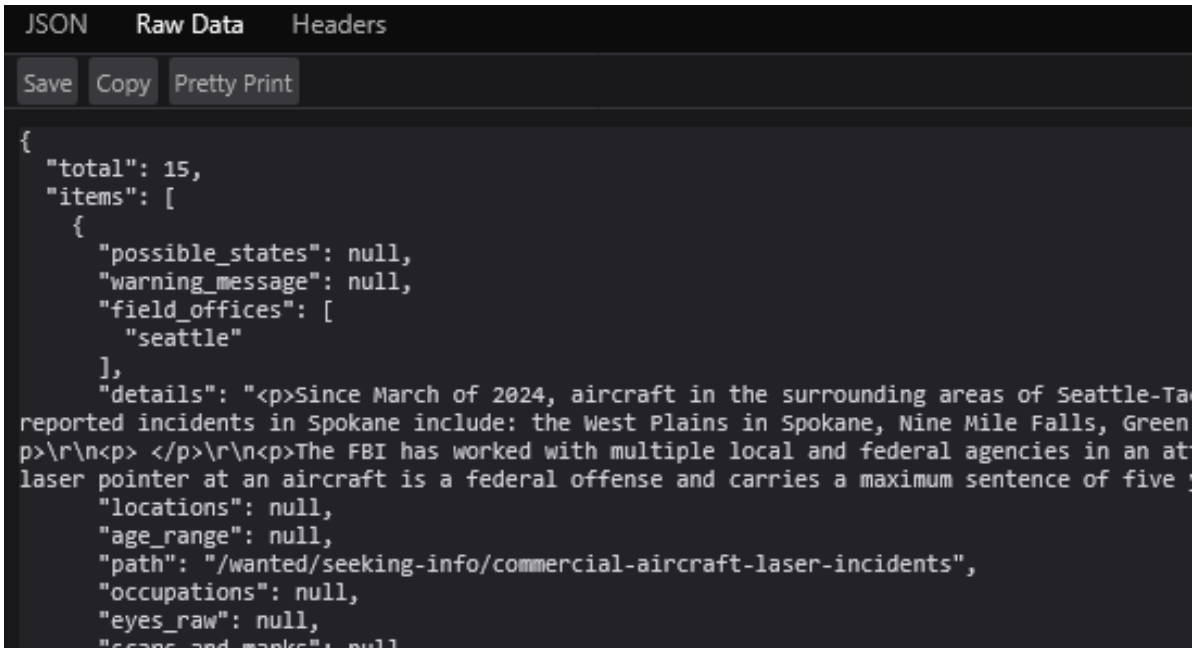
It is indexed with `2`, which means that it is the third element in that array, since indices start with 0. This object contains many key-value pairs, for instance `age_range: null`, or `eyes_raw:"Brown"`, or a longer text as a value of the key `scars_and_marks` .

Among the key-value pairs is one called `field_offices`.

Its values are grouped in an array.

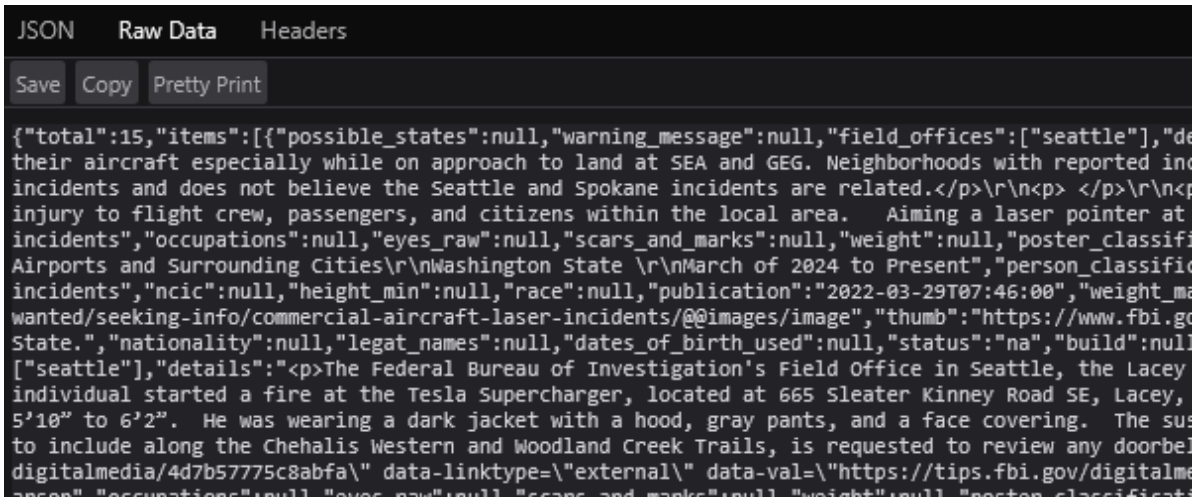
We would need to know the data much better to be able to tell which values are open texts and which are closed lists (and which values comprise them).

10 Raw JSON file - indented



The previous slide was a html rendering of the actual JSON. In this slide you can see it in its raw plain text form. Objects and arrays are indented. This is not obligatory, but without indentation it is very difficult to read for humans. Note that the numeric index of the object was not an actual key, since it is not present in the plain text JSON. There would have to be a key called `0` because it would be the first object. The objects actually have no names.

11 Raw JSON file - no indentation



12 API request with `httr::GET` in R

- you want the content but get more

```
request <- httr::GET("https://api.fbi.gov/wanted/list")
class(request); mode(request); names(request)
```

```
[1] "response"
```

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

```
 [1] "url"         "status_code" "headers"     "all_headers" "cookies"
 [6] "content"     "date"        "times"       "request"     "handle"
```

What comes back is basically a named list. Most elements are metadata. The content of the response is in the `content` element. But it is encoded in raw bytes. `httr` has a function that extracts the content element from the response and decodes it from the raw bytes into characters.

API communications libraries: `RCurl` (old but works), tidyverse's `httr` and `httr2`. To be really proficient in communication with APIs, you need to know things about APIs and internet protocols beyond R. But when the API does not require authentication and you do not need to harvest a lot of data, you will often be fine with just `httr::GET`. Additional parameters

and functions: what to do when the API does not respond instantly, how tell the API that you are not an offender, etc. You can also POST your data into a service as a new input for them to store or use, for instance you can feed your personal details into web forms, or text files to a linguistic parser!

13 Extract content from response

```
req_content <- request %>% httr::content()
mode(req_content); names(req_content)
```

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

```
[1] "total" "items" "page"
```

By default, the `httr::content` function tries to parse the content element from the response object into an R object (typically a list).

This corresponds to the argument value `as = "parsed"`.

Strangely, when you say this explicitly, it is less robust than if you just leave the default. It fails more easily. When it fails (either way), you get back a raw vector.

When you get back a raw vector, choose `as = "text"`. This gives you the original JSON and you can use a different library (`jsonlite`) to parse it into an R object. You can also do this when you are not happy with how `httr::content` parses the JSON. `jsonlite` gives you more control over the JSON parsing.

14 Extract content as text

- when you are not happy with the direct parsing

```
text <-  httr::GET("https://api.fbi.gov/wanted/v1/list?page=1") %>%
  httr::content(as = "text")
text %>% stringr::str_trunc(width = 350, ellipsis = ".....")
```

```
[1] "{\"total\":1060,\"items\":[{\"possible_states\":null,\"warning_message\":null,\"field_o
```

This is an alternative way of reading the API output. You also end up with an R object, but it may have a different structure because the JSON may be interpreted partly differently if you use a specialized library instead of `httr::content`.

The `as = "text"` parameter will just translate the API output from raw bytes to characters. What you see is a one-element character vector. You can either save it as a text file or load it to an R object with a library that can parse that structure. In case of JSON it is, for instance, `jsonlite`.

15 R object from JSON (text)

- `jsonlite::fromJSON`

```
text_fromjson <- httr::GET("https://api.fbi.gov/wanted/v1/list?page=1") %>%
  httr::content(as = "text") %>% jsonlite::fromJSON()
mode(text_fromjson); class(text_fromjson); names(text_fromjson)
```

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

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

```
[1] "total" "items" "page"
```

The `jsonlite::fromJSON` function has many arguments, with sensible defaults. Check out its documentation if you want to play with how exactly the resulting R object should be structured.

16 Structure of FBI Wanted `items`

```
names(req_content)
```

```
[1] "total" "items" "page"
```

```
req_content$total # total count of items (all pages)
```

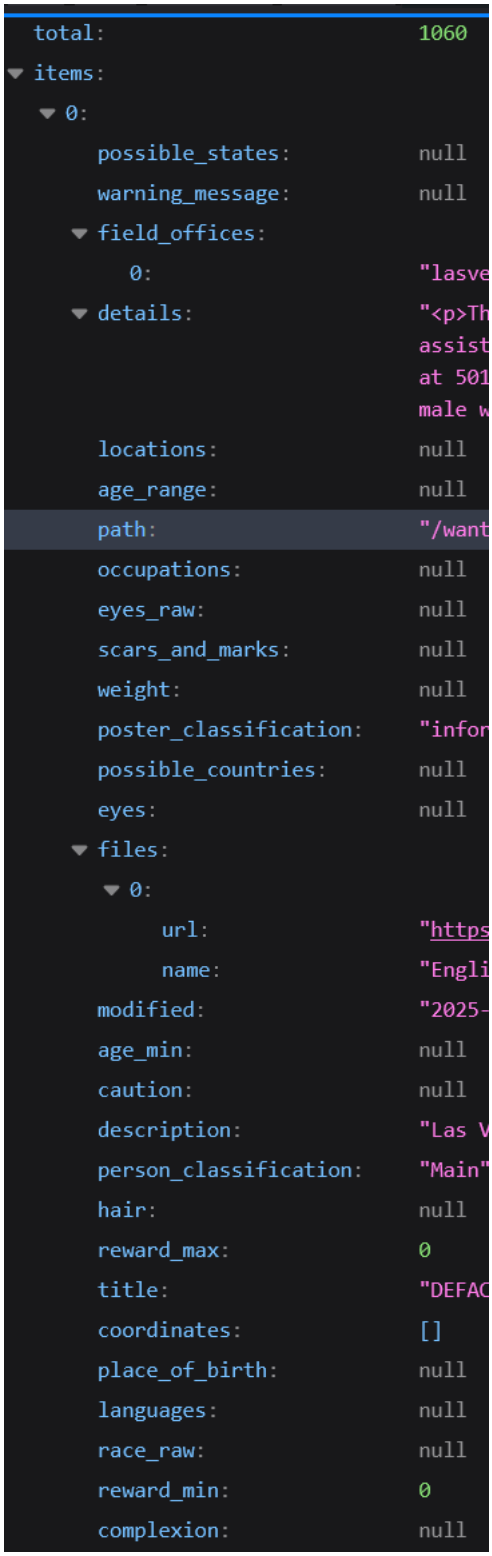
```
[1] 1060
```

```
    req_content$page # which page
```

```
[1] 1
```

```
    length(req_content$items) # count of items on page
```

```
[1] 20
```


total: 1060
items:
0:
possible_states: null
warning_message: null
field_offices:
0: "lasve
details: "<p>Th
assist
at 501
male w
locations: null
age_range: null
path: "/want
occupations: null
eyes_raw: null
scars_and_marks: null
weight: null
poster_classification: "infor
possible_countries: null
eyes: null
files:
0:
url: "https
name: "Engli
modified: "2025-
age_min: null
caution: null
description: "Las V
person_classification: "Main"
hair: null
reward_max: 0
title: "DEFAC
coordinates: []
place_of_birth: null
languages: null
race_raw: null
reward_min: 0
complexion: null

Now we zoom in back on `items`. It has become the third element of a list that we store in the variable `req_content`. The first and the second element store just one value each, whereas the `items` element has a complex internal structure. We can either treat it as a list and access it as a list, or we access it as a data frame with *nested columns*.

17 Zoom in on `items`

```
items <- req_content$items
class(items); names(items); length(items)
```

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

```
NULL
```

```
[1] 20
```

18 Element names in `items`

```
names(items[[1]]) # one item
```

```
 [1] "possible_states"        "warning_message"        "field_offices"
 [4] "details"                "locations"              "age_range"
 [7] "path"                   "occupations"            "eyes_raw"
[10] "scars_and_marks"        "weight"                 "poster_classification"
[13] "possible_countries"     "eyes"                   "files"
[16] "modified"               "age_min"                "caution"
[19] "description"            "person_classification"  "hair"
[22] "reward_max"             "title"                  "coordinates"
[25] "place_of_birth"         "languages"              "race_raw"
[28] "reward_min"             "complexion"             "aliases"
[31] "url"                    "ncic"                   "height_min"
[34] "race"                   "publication"            "weight_max"
[37] "subjects"               "images"                 "suspects"
[40] "remarks"                "reward_text"            "nationality"
[43] "legat_names"            "dates_of_birth_used"    "status"
[46] "build"                  "weight_min"             "hair_raw"
[49] "uid"                    "sex"                    "height_max"
[52] "additional_information" "age_max"                "pathId"
```

We assume that all items have the same structure (they indeed do). To search and filter this list, you would have to write many loops checking the values of the named elements within each of the twenty items. Or you would have to learn to use the `purrr` library. Life would be much easier if the items object was a data frame with twenty rows (items) and fifty-four columns (names of the elements within each item).

19 Get `items` as a data frame

- parse the JSON with `jsonlite`
- make structure as flat as possible (almost default)

```
class(text_fromjson); names(text_fromjson)
```

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

```
[1] "total" "items" "page"
```

20 `items` as a data frame

```
class(text_fromjson$items); colnames(text_fromjson$items)
```

```
[1] "data.frame"
```

```
 [1] "possible_states"        "warning_message"        "field_offices"
 [4] "details"                "locations"              "age_range"
 [7] "path"                   "occupations"            "eyes_raw"
[10] "scars_and_marks"        "weight"                 "poster_classification"
[13] "possible_countries"     "eyes"                   "files"
[16] "modified"               "age_min"                "caution"
[19] "description"            "person_classification"  "hair"
[22] "reward_max"             "title"                  "coordinates"
[25] "place_of_birth"         "languages"              "race_raw"
[28] "reward_min"             "complexion"             "aliases"
[31] "url"                    "ncic"                   "height_min"
[34] "race"                   "publication"            "weight_max"
[37] "subjects"               "images"                 "suspects"
```

```
[40] "remarks"                "reward_text"            "nationality"
[43] "legat_names"            "dates_of_birth_used"    "status"
[46] "build"                  "weight_min"             "hair_raw"
[49] "uid"                    "sex"                    "height_max"
[52] "additional_information" "age_max"                "pathId"
```

21 Nested dataframe `items_df`

```
items_df <- text_fromjson$items
nonatomic <- items_df %>% select(where(~ !is.atomic(.x)))
colnames(nonatomic)
```

```
[1] "field_offices"       "occupations"         "files"
[4] "coordinates"         "languages"           "aliases"
[7] "subjects"            "images"              "dates_of_birth_used"
```

Normally, a data frame column is a vector of values ("atomic"). We suspect that the non-atomic ones are lists, but we check it in the next slide.

22 What class are the non-atomic columns?

```
for (i in seq_along(colnames(nonatomic))) {
  nonatomic %>% pull(i) %>% class() %>% print()
}
```

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

23 Use purrr to avoid loops

```
items_df %>% keep(~ !is.atomic(.x)) %>% map_chr(~ class(.x))
```

```
  field_offices     occupations           files     coordinates
         "list"          "list"          "list"          "list"
      languages         aliases        subjects          images
         "list"          "list"          "list"          "list"
dates_of_birth_used
         "list"
```

24 What is inside the lists?

```
items_df %>%
  purrr::keep(~ !is.atomic(.x)) %>%
  glimpse()
```

```
Rows: 20
Columns: 9
$ field_offices       <list> "lasvegas", "louisville", <NULL>, "miami", "dalla~
$ occupations         <list> <NULL>, <NULL>, <NULL>, <NULL>, <NULL>, <NULL>, <~
$ files               <list> [<data.frame[1 x 2]>], [<data.frame[1 x 2]>], [<d~
$ coordinates         <list> [], [], [], [], [], [], [], [], [], [], [], [~
$ languages           <list> <NULL>, <NULL>, <NULL>, <NULL>, <NULL>, <"English~
$ aliases             <list> <NULL>, <NULL>, <NULL>, <NULL>, <"Cecilia Rodrigu~
$ subjects            <list> "Seeking Information", "Criminal Enterprise Inves~
$ images              <list> [<data.frame[5 x 4]>], [<data.frame[2 x 4]>], [<d~
$ dates_of_birth_used <list> <NULL>, "September 25, 1980", <NULL>, "March 27, ~
```

What is inside the columns that are lists, when it is not `NULL`? For instance, `field_offices` accommodates a character vector on each row. (The first six are single values, but then you see `<"albuquerque", "phoenix">`.)

`files` accommodates a data frame on each row. You can see that they differ in the number of rows but all appear to have two columns.

`coordinates` accommodates a nested list on each row.

So much we see, the elements of the list columns are of the same class, but they have different structures.

25 Zoom in on nested columns

```
(langs_df <- items_df %>%
    select(title, languages, files, field_offices,
      dates_of_birth_used) %>%
  filter(map_lgl(languages, ~!is.null(.x))) ) %>%
  glimpse()
```

```
Rows: 2
Columns: 5
$ title               <chr> "ELLA MAE BEGAY", "MIGUEL ANGEL AGUILAR OJEDA"
$ languages           <list> <"English", "Navajo">, <"Spanish", "English">
$ files               <list> [<data.frame[3 x 2]>], [<data.frame[2 x 2]>]
$ field_offices       <list> <"albuquerque", "phoenix">, "philadelphia"
$ dates_of_birth_used <list> "September 28, 1958", "August 5, 1990"
```

We will filter rows where the `languages` column is not `NULL`. Luckily, even some of their other columns are nested, so we can observe the data unfold in several columns.

26 `tidyr::unnest`

- `unnest`,
- `unnest_longer`,
- `unnest_wider`

You can split the elements of the nested column(s) into rows and/or columns. With nested data frames you may need both directions.

27 `tidyr::unnest_longer`

```
langs_df %>%
  unnest_longer(languages) %>%
  glimpse()
```

```
Rows: 4
Columns: 5
$ title               <chr> "ELLA MAE BEGAY", "ELLA MAE BEGAY", "MIGUEL ANGEL ~
$ languages           <chr> "English", "Navajo", "Spanish", "English"
$ files               <list> [<data.frame[3 x 2]>], [<data.frame[3 x 2]>], [<da~
$ field_offices       <list> <"albuquerque", "phoenix">, <"albuquerque", "phoe~
$ dates_of_birth_used <list> "September 28, 1958", "September 28, 1958", "Augu~
```

Preferable when the numbers of elements differ (e.g. 1 - 10 languages). Each language will split on one row, replicating all other columns unchanged. Like `tidyr::separate_longer`.

After unnesting, Ella Mae Begay will be on two rows (English and Navajo) and Miguel Angel Aguilar Ojeda will also be on two rows (English and Spanish).

28 `tidyr::unnest_wider`

```
langs_df %>%
  unnest_wider(languages, names_sep = "--") %>%
  glimpse()
```

```
Rows: 2
Columns: 6
$ title               <chr> "ELLA MAE BEGAY", "MIGUEL ANGEL AGUILAR OJEDA"
$ `languages--1`      <chr> "English", "Spanish"
$ `languages--2`      <chr> "Navajo", "English"
$ files               <list> [<data.frame[3 x 2]>], [<data.frame[2 x 2]>]
$ field_offices       <list> <"albuquerque", "phoenix">, "philadelphia"
$ dates_of_birth_used <list> "September 28, 1958", "August 5, 1990"
```

Like `tidyr::separate_wider`. Each element splits into a new column. So, Ella and Miguel have each their one row, with two new columns for languages named `languages--1` and `languages--2`. The function provides arguments to automatically generate column names in many ways (we will not cover this here). Using this function can be tricky if the lengths of the arguments differ in rows. If most rows have just one element in this column and a single row has ten elements in this column, you will get ten columns with mostly `NA`s.

29 `unnest_longer` several columns

```
    (un01_df <- langs_df %>%
        unnest_longer(c(field_offices, languages))) %>%
      glimpse()
```

```
Rows: 4
Columns: 5
$ title               <chr> "ELLA MAE BEGAY", "ELLA MAE BEGAY", "MIGUEL ANGEL ~
$ languages           <chr> "English", "Navajo", "Spanish", "English"
$ files               <list> [<data.frame[3 x 2]>], [<data.frame[3 x 2]>], [<da~
$ field_offices       <chr> "albuquerque", "phoenix", "philadelphia", "philad~
$ dates_of_birth_used <list> "September 28, 1958", "September 28, 1958", "Augus~
```

Unnesting is not as powerful as pivoting. Look at this example: posters with Ella Mae Begay were issued in English and in Navajo, in Albuquerque and in Phoenix. The table suggests that the English version is associated with Albuquerque and the Navajo version with Phoenix, but that is random. If you divide the unnesting in two steps, you will get all possible combinations.

30 `unnest_longer` column by column

```
    (un02_df <- langs_df %>%
        unnest_longer(field_offices) %>%
        unnest_longer(languages)) #%>%
```

```
# A tibble: 6 x 5
  title                     languages files  field_offices dates_of_birth_used
  <chr>                     <chr>     <list> <chr>         <list>
1 ELLA MAE BEGAY            English   <df>   albuquerque   <chr [1]>
2 ELLA MAE BEGAY            Navajo    <df>   albuquerque   <chr [1]>
3 ELLA MAE BEGAY            English   <df>   phoenix       <chr [1]>
4 ELLA MAE BEGAY            Navajo    <df>   phoenix       <chr [1]>
5 MIGUEL ANGEL AGUILAR OJEDA Spanish  <df>   philadelphia  <chr [1]>
6 MIGUEL ANGEL AGUILAR OJEDA English  <df>   philadelphia  <chr [1]>
```

```
    # glimpse()
```

When you divide the unnesting into two steps, you will get all possible combinations. So, Ella will be on four rows: English Albuquerque, Navajo Albuquerque, English Phoenix, and Navajo Phoenix, while Miguel will be on two rows: Spanish and English, both with Philadephia.

The original data structure is ambiguous with respect to which field office used which language to advertise the wanted person in a poster.

At least it looks as if they record advertising languages, judged by the fact that the `files` column contains the URLs of file names of the posters in PDF and something called "name", which could be the language, but possibly a title, too. If it mattered in which office they issued or presented which files, they would put this information together into a list or a data frame. Maybe all listed languages were used in all listed places or it really does not matter to FBI and it is false from us to enforce a disambiguation here. It depends on our use case what the ideal structure would be.

31 Unnesting a data frame

```
(un04_df <- langs_df %>%
    select(title, files) %>%
    unnest(files)) %>%
  glimpse()
```

```
Rows: 5
Columns: 3
$ title <chr> "ELLA MAE BEGAY", "ELLA MAE BEGAY", "ELLA MAE BEGAY", "MIGUEL AN~
$ url   <chr> "https://www.fbi.gov/wanted/kidnap/ella-mae-begay/download.pdf",~
$ name  <chr> "English", "DIN BIZAAD KEHGO", "EN ESPAOL", "English", "EN ESPAO~
```

With data frames, it makes sense that they unnest in both dimensions, when the nested data frames have the same number of columns in all rows.

32 Nested data frame with `unnest_wider`

```
(un05_df <- langs_df %>%
    select(title, files) %>%
    unnest_wider(files, names_sep = "---")) %>%
  glimpse()
```

```
Rows: 2
Columns: 3
$ title          <chr> "ELLA MAE BEGAY", "MIGUEL ANGEL AGUILAR OJEDA"
$ `files---url`  <list<chr>> <"https://www.fbi.gov/wanted/kidnap/ella-mae-begay/down~
$ `files---name` <list<chr>> <"English", "DIN BIZAAD KEHGO", "EN ESPAOL">, <"E~
```

Here you also get two new columns, but the rows will not divide. You will get character vectors in the columns.

33 Lists in nested columns

```
    langs_FilesAsList %>% pull(files)
```

```
[[1]]
[[1]]$url
[1] "https://www.fbi.gov/wanted/kidnap/ella-mae-begay/download.pdf"
[2] "https://www.fbi.gov/wanted/kidnap/ella-mae-begay/ella-mae-begay_navajo.pdf/@@download/fi
[3] "https://www.fbi.gov/wanted/kidnap/ella-mae-begay/begayspanish.pdf/@@download/file/BegayS

[[1]]$name
[1] "English"          "DIN BIZAAD KEHGO" "EN ESPAOL"


[[2]]
[[2]]$url
[1] "https://www.fbi.gov/wanted/murders/miguel-angel-aguilar-ojeda/download.pdf"
[2] "https://www.fbi.gov/wanted/murders/miguel-angel-aguilar-ojeda/ojedaspanish2.pdf/@@downlo

[[2]]$name
[1] "English"   "EN ESPAOL"
```

Our slice of the FBI Wanted data does not contain any lists in values. Therefore I created a derived data frame `langs_FilesAsList`, where I deliberately transform the `files` column from a list of data frames into a list of lists. Now each cell contains a named list with two elements, `url` and `name`. The next slide will show how to extract information from such lists with a one-liner.

34 `hoist` nested list elements into columns

```
    langs_FilesAsList %>%
     tidyr::hoist(files, POSTER_URL = list( 1L),
       POSTER_NAMES = list(2L))
```

```
                       title        languages
1             ELLA MAE BEGAY  English, Navajo
2 MIGUEL ANGEL AGUILAR OJEDA Spanish, English

1 https://www.fbi.gov/wanted/kidnap/ella-mae-begay/download.pdf, https://www.fbi.gov/wanted/k
2                                                                                           N
                          POSTER_NAMES        field_offices dates_of_birth_used
1 English, DIN BIZAAD KEHGO, EN ESPAOL albuquerque, phoenix  September 28, 1958
2                   English, EN ESPAOL         philadelphia      August 5, 1990
```

With `tidyr::hoist` you can access a specific element of a nested list in a nested column, which itself always is a list.

35 `hoist`, then `unnest`.

```
   langs_FilesAsList %>%
    tidyr::hoist(files, POSTER_URL = list( 1L), POSTER_NAMES = list(2L)) %>%
     unnest_longer(c(starts_with("POSTER")))
```

```
# A tibble: 5 x 6
  title      languages POSTER_URL POSTER_NAMES field_offices dates_of_birth_used
  <chr>      <list>    <chr>      <chr>        <list>        <list>
1 ELLA MAE ~ <chr [2]> https://w~ English      <chr [2]>     <chr [1]>
2 ELLA MAE ~ <chr [2]> https://w~ DIN BIZAAD ~ <chr [2]>     <chr [1]>
3 ELLA MAE ~ <chr [2]> https://w~ EN ESPAOL    <chr [2]>     <chr [1]>
4 MIGUEL AN~ <chr [2]> https://w~ English      <chr [1]>     <chr [1]>
5 MIGUEL AN~ <chr [2]> https://w~ EN ESPAOL    <chr [1]>     <chr [1]>
```

The nested list elements were vectors, so the hoisted columns are still nested columns. But this time they store only vectors, and these are easily extracted with `unnest`.

This is actually the end of this use case.

In the next slide, I will return to how I transformed data frames to lists in a nested columns, because I have used `map`, a very versatile function from the `purrr` library.

36 Do anything inside nested columns

- operations on lists in general: `purrr`
- `purrr::map(<your list>, ~ <some function on your list>(.x, <other arguments>))`

```
langs_FilesAsList <- langs_df %>%
  mutate(across(files, ~ map(.x, ~ as.list(.x))))
```

Our slice of the FBI Wanted data does not contain any lists in values. Therefore I create a derived data frame `langs_FilesAsList`, where I deliberately transform the `files` column from a list of data frames into a list of lists. Now each cell contains a named list with two elements, `url` and `name`. This code contains the `purrr::map` function, which takes each element of a list (here the `files` column) and performs a function on each of its elements (transforms each data frame into a list). You could of course achieve the same thing with a loop, as you will see in the next slide.

37 Transform the dataframes column with a loop

```
test_df <- langs_df
for (i in 1:nrow(langs_df)) {
  test_df$files[[i]] <- as.list(test_df$files[[i]])
}

test_df$files[[1]]
```

```
$url
[1] "https://www.fbi.gov/wanted/kidnap/ella-mae-begay/download.pdf"
[2] "https://www.fbi.gov/wanted/kidnap/ella-mae-begay/ella-mae-begay_navajo.pdf/@@download/fi
[3] "https://www.fbi.gov/wanted/kidnap/ella-mae-begay/begayspanish.pdf/@@download/file/Begay$

$name
[1] "English"          "DIN BIZAAD KEHGO" "EN ESPAOL"
```

This loop edits the elements of `files`, the list column, one by one. Because it is a list, it does not mind accommodating elements of different data classes. So, in each cycle, the list column contains one more two-element list instead of a two-column data frame. (Vectors obviously would not allow you to do something similar, e.g. change the elements from characters to digits one by one!)