Extracting data from the web

APIs and beyond

Scott Chamberlain
Karthik Ram
Garrett Grolemund

June 2016

USER 2016 CONFERENCE

@sckottie

Outline

Part 1 - Collecting data from an API

Scott Chamberlain

Co-Founder

ROpenSci

Outline

Part 1 - Collecting data from an API

Part 2 - Wrapping an API with R

Karthik Ram
Data Scientist and
Ecologist and co-founder
ROpenSci

Outline

Part 1 - Collecting data from an API

Part 2 - Wrapping an API with R

Part 3 - Scraping data without an API

Garrett Grolemund
Master Instructor and
Data Scientist
RStudio®

Your Turn

Introduce yourself to the people at your table.

Determine who among you has the most web development experience.

APIs
Intro

@sckott / @ropensci / @pdxrlang

Outline

1. What is an API?
2. HTTP
3. HTTP verbs
4. HTTP structure
5. Data formats
6. Wrap up

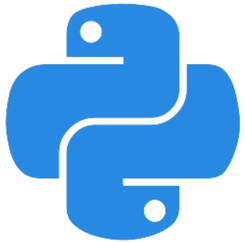
What is an API?

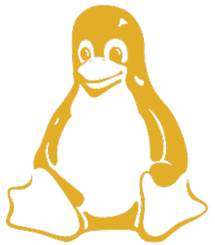
An API is…

Programmatic instructions for how to interact with a piece of software

Can be the interface to:

- A software package in R/Python/etc.
- A public web API
- A database
- An operating system





Most APIs are REST APIs

REST? WTF?

Representational **S**tate **T**ransfer

an architectural style in which most web APIs are constructed

https://en.wikipedia.org/wiki/Representational_state_transfer

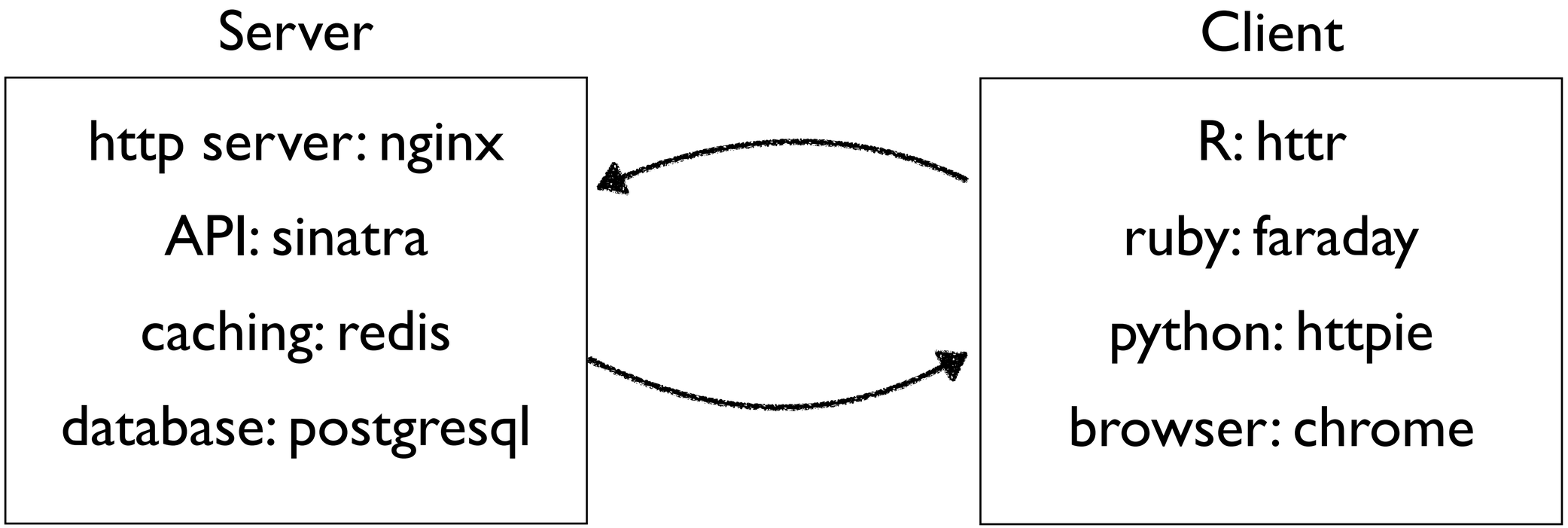
HTTP

Hyper**T**ext **T**ransfer **P**rotocol

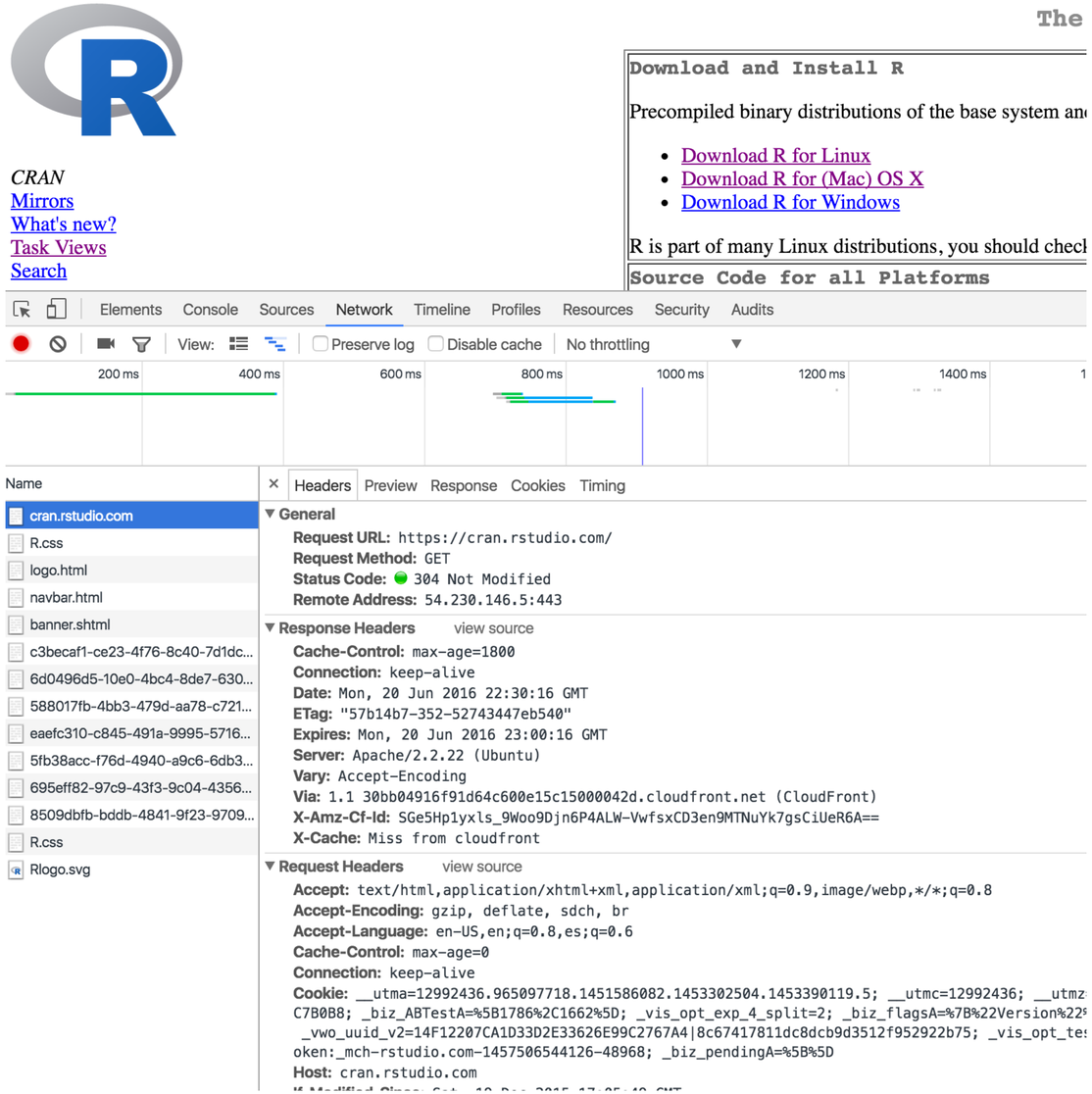
HTTP spec: https://tools.ietf.org/html/rfc7235

- Verbs for different actions
- Authentication
- Status codes
- Request and response format
- Most REST APIs use HTTP for data transfer

But, what does it all look like?



HTTP is behind the scenes



HTTP in R

You’ve been using HTTP in R - For example:

- install.packages() -> uses download.file() under the hood -> which uses http

Your Turn

httr hello world

- Load **httr**
- Use **httr::GET()** to get data from any website.
    - Poke around at the resulting object.
    - Find the *headers*, the *status code*, and the *content*

```
library(httr)
x <- GET('https://google.com/')

x$status_code
#>[1] 200

x$headers
#> $date
#> [1] "Thu, 23 Jun 2016 23:05:27 GMT”
#> ...

x$content
#>[1] 3c 21 64 6f 63 74 79 70 65 20 68 ...
```

HTTP Verbs & Requests

HTTP Verbs

HTTP Verbs

GET	Retrieve whatever is specified by the URL
POST	Create resource at URL with given data
PUT	Update resource at URL with given data
DELETE	Delete resource at URL

HTTP Verbs: GET

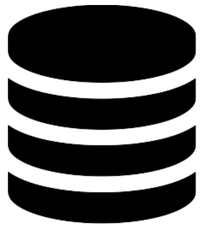
GET https://api.github.com/repos/hadley/dplyr/issues?per_page=3

base url

path

query parameters

send to GitHub's servers



GitHub sends back data!

HTTP Verbs: POST

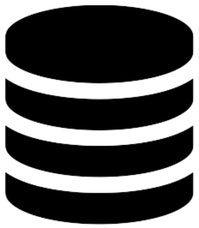
POST https://api.github.com/repos/hadley/dplyr/issues

base url

path

body

```
{
  "title": "Found a bug",
  "body": "I'm having a problem with this.",
  "assignee": "wch",
  "milestone": 2,
  "labels": [
    "bug"
  ]
}
```



HTTP Verbs: PUT

issue #

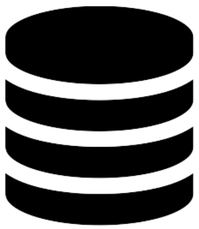
PUT https://api.github.com/repos/hadley/dplyr/issues/3

base url

path

```
{
  "title": "Found a bug",
  "body": "I'm having a problem with this.",
  "assignee": "wch"
}
```

body

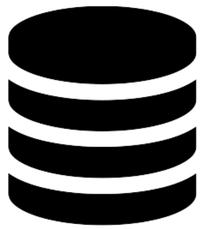


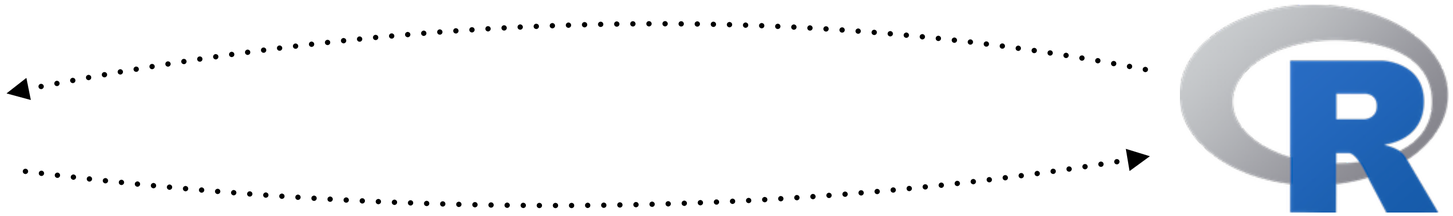
HTTP Verbs: DELETE

DELETE https://api.github.com/repos/sckott/foobar

base url

path





more HTTP Verbs

- HEAD - identical to GET, but just gets headers back
- PATCH - similar to PUT, but partially modify
- COPY - copy a resource from one URI to another
- OPTIONS - get what verbs supported for a URI
- a few others: TRACE, CONNECT

Assembling Queries

HTTP request components

- **URL** - where on the web do you want to make the request, including parameter values
- **Method** - what HTTP verb
- **Headers** - any metadata to modify the request
- **Body** - the data, very flexible, containing strings, files, binary, etc.

Assembling Queries: in R

URL

```
http://…
e.g., GET(url = “http://xxx”)
```

Headers

```
httr::add_headers(hello = “world”)
```

Method

```
httr::GET()
httr::POST()
httr::PUT()
httr::DELETE()
…
```

Body

```
httr::POST(body = list(foo = “bar”))
```

httpbin.org

httpbin(1): HTTP Request & Response Service

Freely hosted in HTTP, HTTPS & EU flavors by Runscope

ENDPOINTS

/ This page.
/ip Returns Origin IP.
/user-agent Returns user-agent.
/headers Returns header dict.
/get Returns GET data.
/post Returns POST data.
/patch Returns PATCH data.
/put Returns PUT data.
/delete Returns DELETE data
/encoding/utf8 Returns page containing UTF-8 data.
/gzip Returns gzip-encoded data.
/deflate Returns deflate-encoded data.
/status/:code Returns given HTTP Status code.
/response-headers?key=val Returns given response headers.
/redirect/:n 302 Redirects *n* times.
/redirect-to?url=foo 302 Redirects to the *foo* URL.
/relative-redirect/:n 302 Relative redirects *n* times.
/absolute-redirect/:n 302 Absolute redirects *n* times.
/cookies Returns cookie data.

Your Turn

httr verbs practice

- **GET** request to https://httpbin.org/get
- **POST** request to https://httpbin.org/post
- Try mismatching a httr method with a httpbin URL, what happens?

Request Components

- Send a request with query parameters
- Send a request with a header
- Send a request with a body

```
library(httr)

GET("https://httpbin.org/get")

POST("https://httpbin.org/post")

x <- POST("https://httpbin.org/get")
x$status_code
#>[1] 405
METHOD NOT ALLOWED!!!!
```

```
library(httr)

# Request with query parameters
x <- GET(url, query = list(a = 5))

# Request with headers
x <- GET(url, add_headers(wave = "hi"))

# Request with a body
x <- POST(url, body = list(a = 5))
```

HTTP Responses

HTTP response components

- **status** - status of the response
- **headers** - response headers, like content type, size of body, paging info, rate limit info, etc.
- **body/content** - many different types, compressed or not, binary or not, etc.

status

- 3 digit numeric code
- One of 5 different classes of codes:
  - **1xx**: informational
  - **2xx**: success
  - **3xx**: redirection
  - **4xx**: client error
  - **5xx**: server error

- Info on status codes: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
- In R: https://cran.rstudio.com/web/packages/httpcode/ for HTTP status code look up

status: beware

- Servers do not always give correct codes
- Clients may pass on these inappropriate codes
- i.e., Don’t trust status codes alone - use in combination with other information:
  - content type
  - body length
  - etc.

Your Turn

Look up different status codes by using

https://http.cat/<HTTP STATUS CODE>

418: “I’m a teapot”

https://http.cat/418

headers

- Contain metadata about the **Request** & **Response**
- Some headers standardized
- Some headers custom for the web service
- Most headers **key:value** pairs
- Some headers just **value** without a key

headers

http://httpbin.org/get

request

```
GET /get HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: httpbin.org
User-Agent: HTTPie/0.9.2
```

response

```
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 228
Content-Type: application/json
Date: Wed, 22 Jun 2016 16:12:04 GMT
Server: nginx
```

content / body

```
x <- GET('https://google.com/')
x$content
#>[1] 3c 21 64 6f 63 74 79 70 65 20 68 …

content(x)
```

raw bytes

to extract data

More in Part II

Your Turn

Using http://httpbin.org/get

- Get status code from an httr response object - Use httr to figure out what the code means
- From a http response: Get request & response headers -> Then extract content type
- Change the request content type - i.e,. the accept content type

Using http://httpbin.org/status/<status code>

- Do request for each of 400, and 500 - what do you get for content()?

```
library(httr)

res <- GET(“http://httpbin.org/get")

# status code
code <- res$status_code
http_status(code) # or http_status(res)

# content type
res$request$headers[[1]]
res$headers$`content-type`

# change accept content type
res <- GET(“http://httpbin.org/get”, accept_json())
```

```
library(httr)

# status code: 400
res <- GET("http://httpbin.org/status/400")
res
#> [1] NULL

# status code: 500
res <- GET("http://httpbin.org/status/500")
res
#> [1] NULL



# the content isn't always empty!  Look in content AND
headers for error messages
```

Data Formats

JSON

http://www.omdbapi.com/?t=frozen&y=&plot=short&r=json


```
{
  "Title": "Frozen",
  "Year": "2013",
  "Rated": "PG",
  "Released": "27 Nov 2013",
  "Runtime": "102 min",
  "Genre": "Animation, Adventure, Comedy",
  "Director": "Chris Buck, Jennifer Lee",
  "Writer": "Jennifer Lee (screenplay), Hans Christian Andersen (story inspired by \"The Snow Queen\" by), Chris Buck (story by), Jennifer Lee (story by), Shane Morris (story by)",
  "Actors": "Kristen Bell, Idina Menzel, Jonathan Groff, Josh Gad",
  "Plot": "When the newly crowned Queen Elsa accidentally uses her power to turn things into ice to curse her home in infinite winter, her sister, Anna, teams up with a mountain man, his playful reindeer, and a snowman to change the weather condition.",
  "Language": "English, Icelandic",
  "Country": "USA",
  "Awards": "Won 2 Oscars. Another 70 wins & 56 nominations.",
  "Poster": "http://ia.media-imdb.com/images/M/MV5BMTQ1MjQwMTE5OF5BMl5BanBnXkFtZTgwNjk3MTcyMDE@._V1_SX300.jpg",
  "Metascore": "74",
  "imdbRating": "7.6",
  "imdbVotes": "410,734",
  "imdbID": "tt2294629",
  "Type": "movie",
  "Response": "True"
}
```

JSON

- **J**avascript **O**bject **N**otation

- Widely used in web APIs
- Becoming de facto standard for data format for web APIs
- less expressive than XML
- but easier for humans to grok
- jsonlite - the go to JSON pkg for R, to create and parse JSON

jsonlite

https://cran.rstudio.com/web/packages/jsonlite

```
library(jsonlite)
fromJSON('{"foo": "bar"}')
#> $foo
#> [1] "bar"

fromJSON('{"foo": "bar"}', FALSE)
#> $foo
#> [1] "bar"

fromJSON('[{"foo": "bar", "hello": "world"}]')
#>   foo hello
#> 1 bar world
```

XML

http://www.omdbapi.com/?t=frozen&y=&plot=short&r=xml


```
<root response="True">
    <movie title="Frozen" year="2013" rated="PG" released="27 Nov
    2013" runtime="102 min" genre="Animation, Adventure, Comedy"
    director="Chris Buck, Jennifer Lee" writer="Jennifer Lee
    (screenplay), Hans Christian Andersen (story inspired by
    &quot;The Snow Queen&quot; by), Chris Buck (story by), Jennifer
    Lee (story by), Shane Morris (story by)" actors="Kristen Bell,
    Idina Menzel, Jonathan Groff, Josh Gad" plot="When the newly
    crowned Queen Elsa accidentally uses her power to turn things
    into ice to curse her home in infinite winter, her sister,
    Anna, teams up with a mountain man, his playful reindeer, and a
    snowman to change the weather condition." language="English,
    Icelandic" country="USA" awards="Won 2 Oscars. Another 70 wins
    & 56 nominations." poster="http://ia.media-imdb.com/images/M/
    MV5BMTQ1MjQwMTE5OF5BMl5BanBnXkFtZTgwNjk3MTcyMDE@._V1_SX300.jpg"
    metascore="74" imdbRating="7.6" imdbVotes="410,734"
    imdbID="tt2294629" type="movie"/>
</root>
```

XML

- E**x**tensible **M**arkup **L**anguage

- Used to dominate in web APIs, no less common
- Very expressive
- hard for humans to grok
- xml2 - the go to XML pkg for R, to create and parse XML

xml2

https://cran.rstudio.com/web/packages/xml2

```
library(xml2)

res <- read_xml('<foo>bar</foo>')
xml_name(res)
#> [1] “foo"

xml_text(res)
#> [1] "bar"
```

Your Turn

Using the IMDB API: http://www.omdbapi.com/

Get data for 3 movies in both JSON and XML format.

Parse each format to plain text and their parsed versions.

```
library(httr)

j1 = GET("http://www.omdbapi.com/?t=iron%20man%202&r=json")

content(j1, as = "text")
content(j1, as = "parsed")



x1 = GET("http://www.omdbapi.com/?t=iron%20man%202&r=xml")

content(x1, as = "text")
content(x1, as = "parsed")
```

Recap

APIs: many components - we focused on HTTP

HTTP verbs: **GET** [?] **POST** [?] **PUT**, **DELETE,** etc.

URL / Methods / Header / Body

HTTP **request**

Status / Headers / Body

HTTP **response**

{"foo": "bar"}
<foo>bar</foo>

Data formats: **JSON** and **XML**

thank you

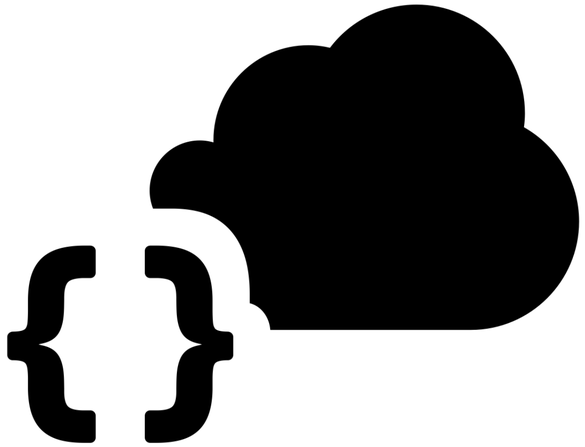
Outline

Part 1 - Collecting data from an API

Part 2 - Wrapping an API with R

Karthik Ram
Data Scientist and
Ecologist and co-founder
ROpenSci

Part II: Getting data from the web



Karthik Ram

@_inundata

1. Making **GET** requests
2. **Extracting web content**
3. Passing **additional query parameters** and **API** keys

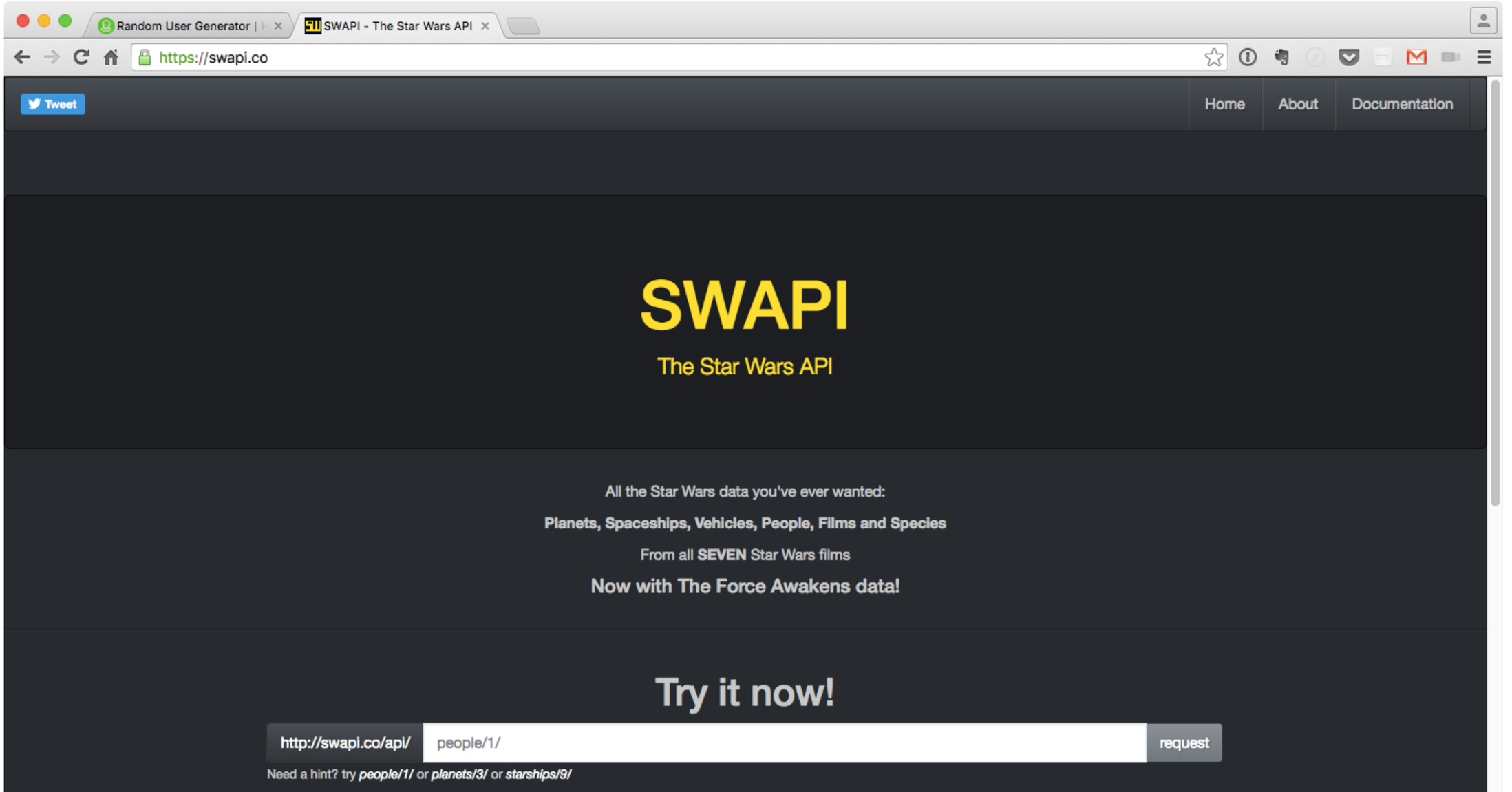
APIs and end points

The Internet Movie database

Producers Movies Actors


End points

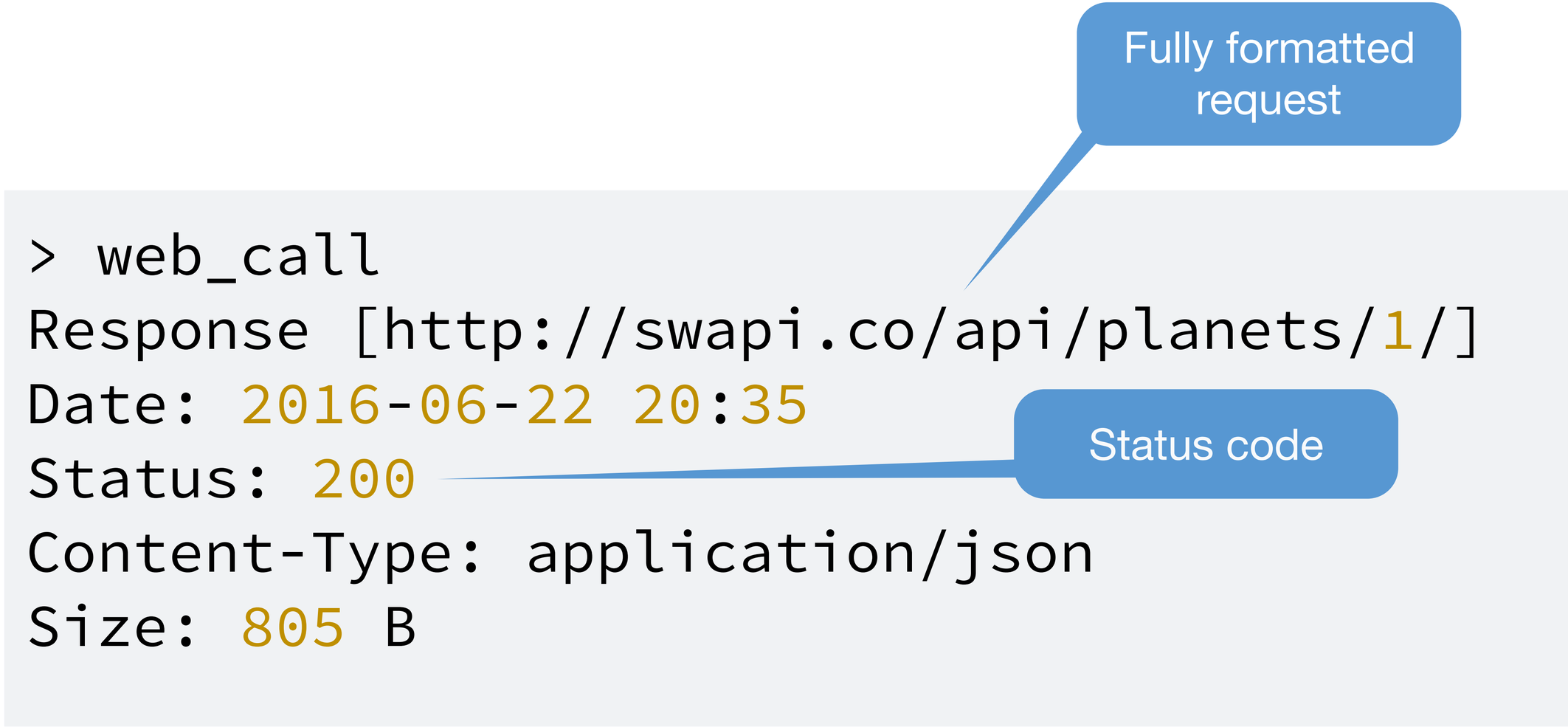
Writing a GET request


Random User Generator |
SWAPI - The Star Wars API
https://swapi.co
Tweet
Home
About
Documentation
SWAPI
The Star Wars API
All the Star Wars data you've ever wanted:
Planets, Spaceships, Vehicles, People, Films and Species
From all SEVEN Star Wars films
Now with The Force Awakens data!
Try it now!
http://swapi.co/api/
people/1/
request
Need a hint? try people/1/ or planets/3/ or starships/9/

Step 1: Write a web request

```
web_call <- GET('http://swapi.co/api/planets/1/')
web_call
```

```
> web_call
Response [http://swapi.co/api/planets/1/]
Date: 2016-06-22 20:35
Status: 200
Content-Type: application/json
Size: 805 B
```


Fully formatted request
> web_call
Response [http://swapi.co/api/planets/1/]
Date: 2016-06-22 20:35
Status: 200
Status code
Content-Type: application/json
Size: 805 B

Now your turn

Run a GET request against these end points

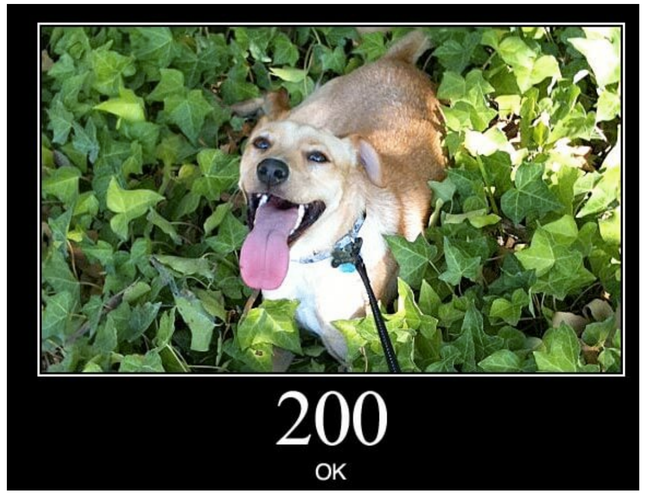
```
api.randomuser.me
api.openweathermap.org/data/2.5/forecast?id=524901
```

What does your result object contain?
What are the response codes?
What do they mean?

```
result <- GET('api.randomuser.me')
status_code(result)

result_2 <- GET('api.openweathermap.org/data/
2.5/forecast?id=524901')
status_code(result_2)
```

See full list of codes: httpstatuses.com



2×× SUCCESS

200 OK

The request has succeeded.

The payload sent in a 200 response depends on the request method.

4×× CLIENT ERROR

403 FORBIDDEN

The server understood the request but refuses to authorize it.

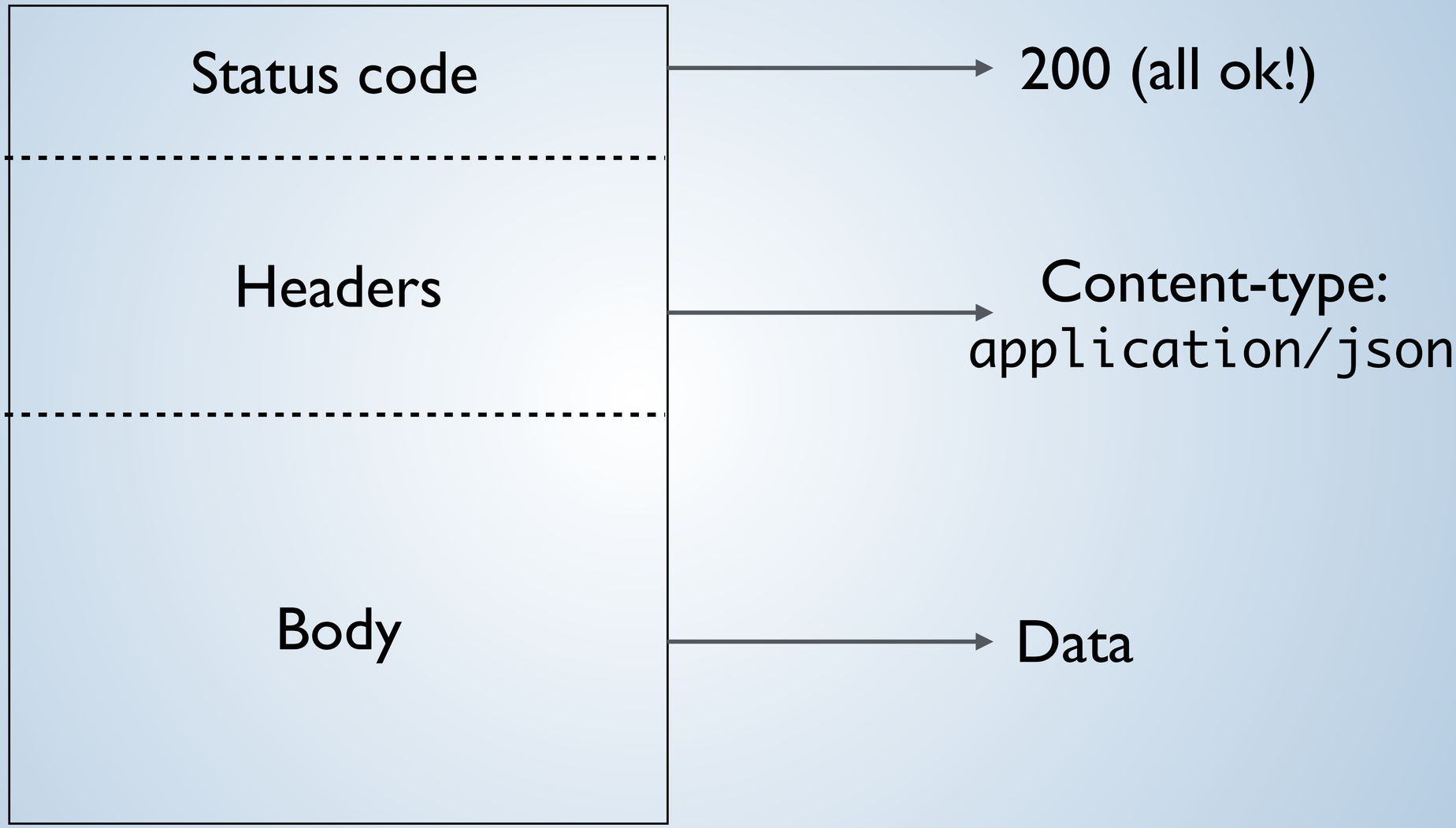
A server that wishes to make public why the request has been forbidden can describe that reason in the response payload (if any).

If authentication credentials were provided in the request, the server considers them insufficient to grant access. The client SHOULD NOT automatically repeat the request with the same credentials. The client MAY repeat the request with new or different credentials. However, a request might be forbidden for reasons unrelated to the credentials.

Pro tip

```
stop_for_status(request)
warn_for_status(request)
```

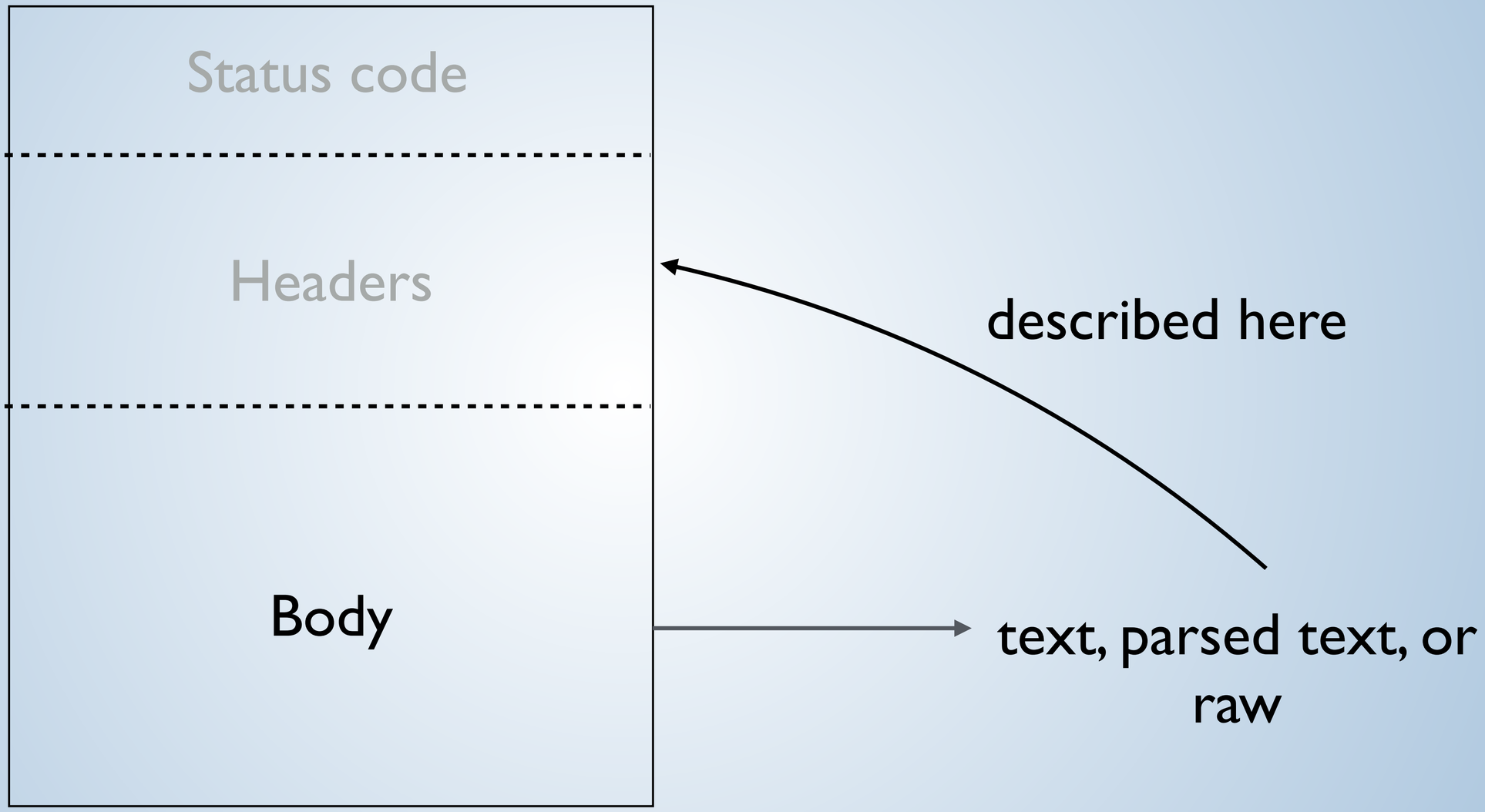
2: Extracting content


Status code
Headers
Body
200 (all ok!)
Content-type:
application/json
Data

Response

Step 2: Extract content from a request

text - *a character vector*
raw - *as a raw object*
parsed - *parsed into a R object*


Status code
Headers
Body
described here
text, parsed text, or raw

Text formats

```
text/html: read_html
text/xml: read_xml
text/csv: read_csv
text/tab-separated-values: read_tsv
application/json: fromJSON
```

Local R handler

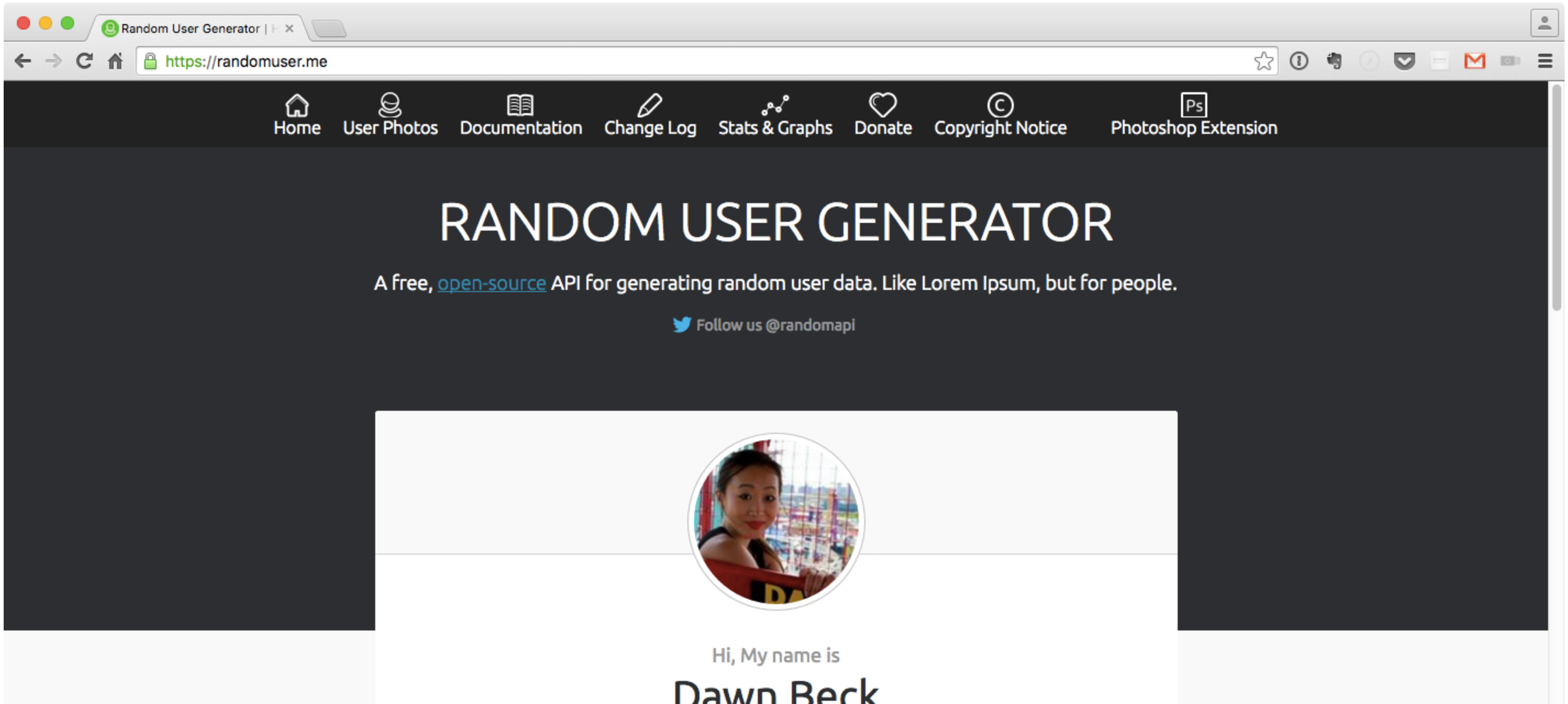
Content type

Image formats

```
image/jpeg: readJPEG
image/png: readPNG
```

```
> call <- GET("http://google.com")
> result <- content(call, as = 'text')
> result
[1] "<!doctype html><html itemscope=\"\" itemtype=\"http://
schema.org/WebPage\" lang=\"en\"><head><meta content=\"Search the
world's information, . <truncated>
> class(result)
[1] "character"
```

api.randomuser.me



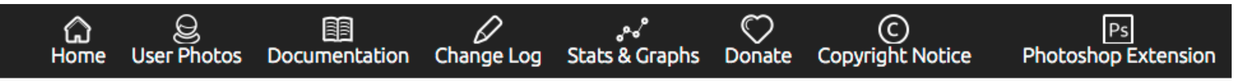
Exercise

Process the content from the random user API

Save the result into an object called `person`

Extract the content as text and parsed.

Passing arguments



Requesting Multiple Users

Random User Generator allows you to fetch up to 5,000 generated users in one request using the **results** parameter.

```
http://api.randomuser.me/?results=5000
```

results: *Retrieve multiple names*

Specifying a gender

You can specify whether you would like to have only male or only female users generated by adding the **gender** parameter to your request. Valid values for the gender parameter are "male" or "female", or you may leave the parameter blank. Any other value will cause the service to return both male and female users.

```
http://api.randomuser.me/?gender=female
```

gender: *specify a gender*

Seeds

Seeds allow you to always generate the same set of users. For example, the seed "foobar" will always return results for Becky Sims (for version 1.0). Seeds can be any string or sequence of characters.

```
http://api.randomuser.me/?seed=foobar
```

seed: *Set a seed*

Formats

We currently offer the following data formats:

```
num_results <- 5
args <- list(results = num_results)
random_names <- GET("http://api.randomuser.me/",
query = args)
output <- content(random_names, as = 'parsed')
> length(output$results)
[1] 5
```

Exercise

Look over the api documentation and see what other parameters can be passed

Modify the example to pass a `gender` to the request

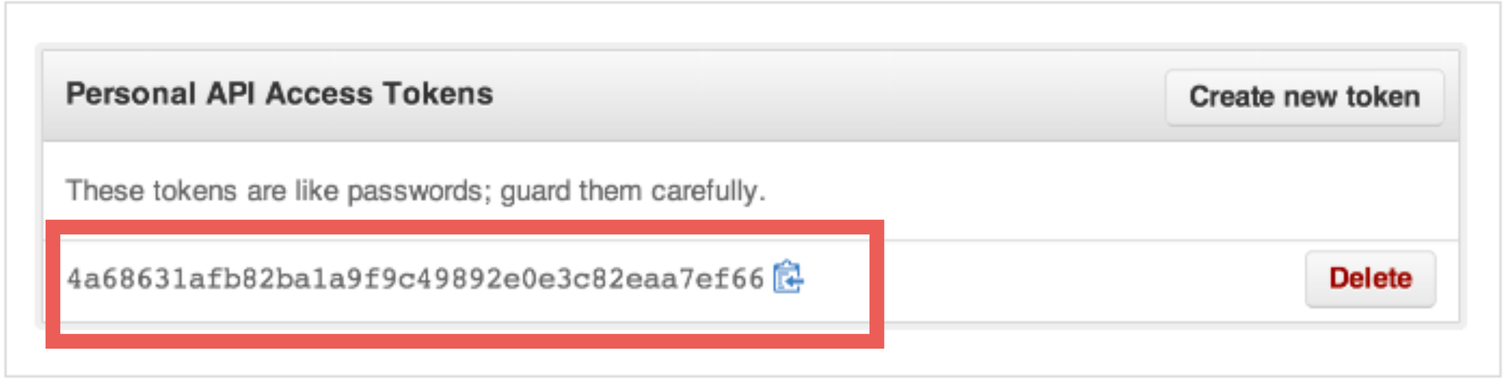
Can you do this with both number of results and gender?

Personal Open source Business Explore Pricing Blog Support Search Gi

Personal API tokens

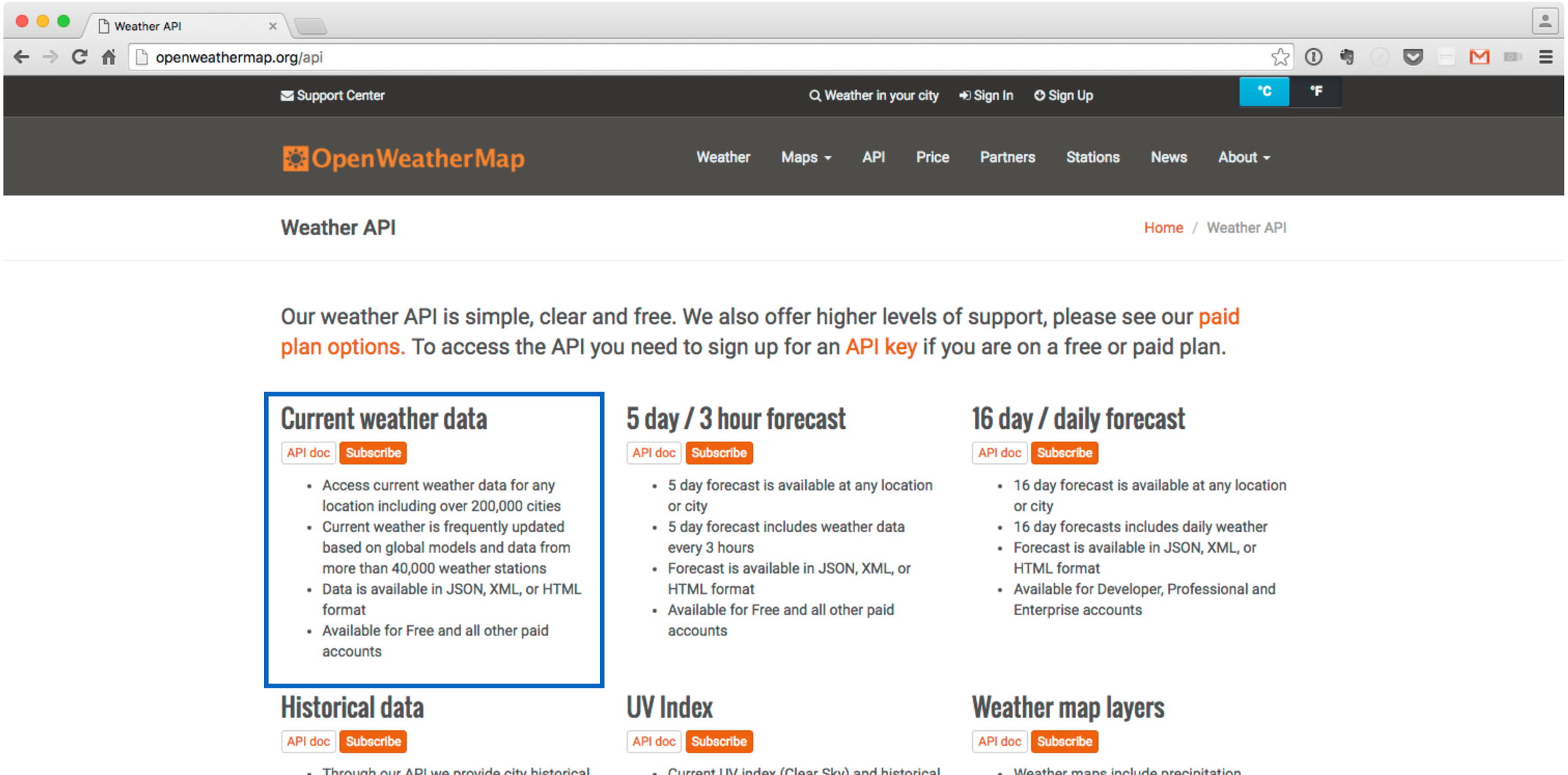
May 16, 2013 tclem New Features

You can now create your own personal API tokens for use in scripts and on the command line. Be careful, these tokens are like passwords so you should guard them carefully. The advantage to using a token over putting your password into a script is that a token can be revoked, and you can generate lots of them. Head on over to your settings to manage personal API tokens.



Write a complete request to a proper API

Open Weather example



Exercise

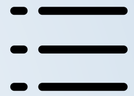
- Get a API key from `openweathermap.org/api`
- Write a request for current weather data (http://openweathermap.org/current) for zip code `94708,us`
- Pass
- Format results into a `data.frame` (for help with this, see: bit.ly/flatten_json)

Tip: Pass the API key as:

```
APPID <- '<your_api_key>'
```

Recap

Make a `GET` request

Pass additional **arguments** to a end point 

Authenticate with a **API** key or `oauth2` **token**

Check for **status**

Then **process content** as 

Format results 

Thanks!