


APIs

Intro

HELLO

my name is

Scott

 @sckott / @ropensci /
@pdxrlang

Outline

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

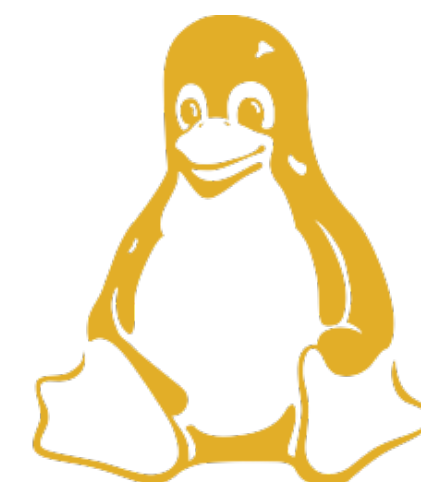
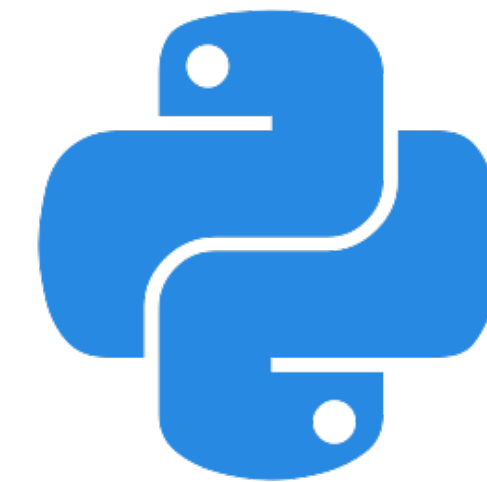
What is an
API?

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](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

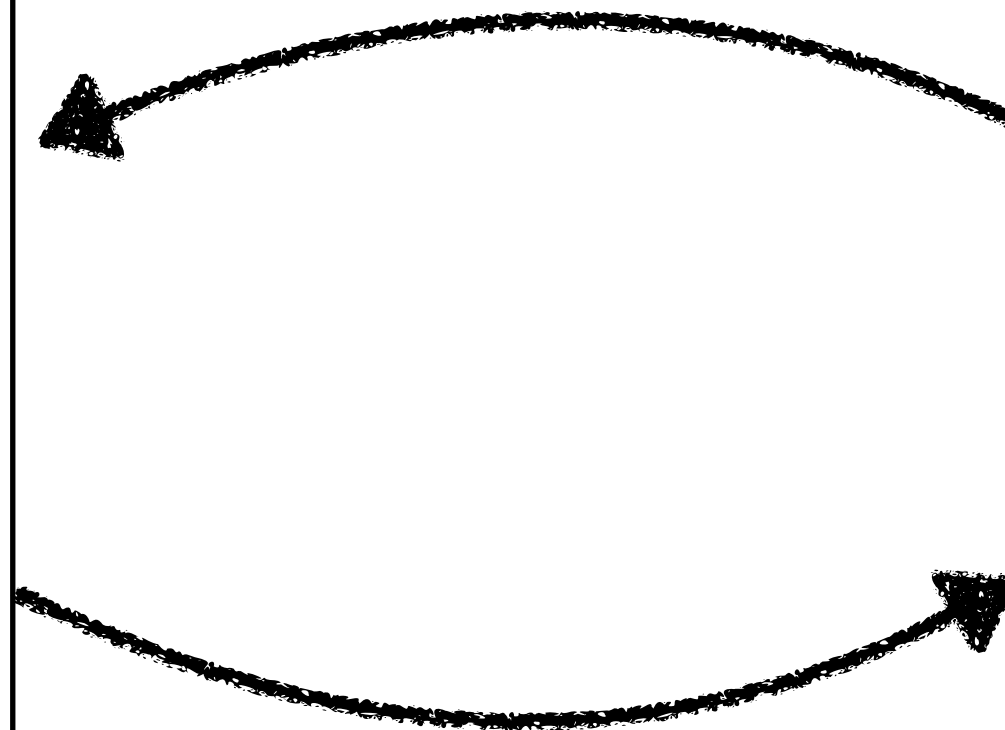
But, what does it all look like?

Server


http server: nginx
API: sinatra
caching: redis
database: postgresql

Client

R: httr
ruby: faraday
python: httpie
browser: chrome



HTTP is behind the scenes



CRAN
[Mirrors](#)
[What's new?](#)
[Task Views](#)
[Search](#)

The

Download and Install R

Precompiled binary distributions of the base system and

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check

Source Code for all Platforms

⌵ ⌵ Elements Console Sources Network Timeline Profiles Resources Security Audits

● 🔇 📺 🔍 View: 📄 📄 ☐ Preserve log ☐ Disable cache No throttling ▼

200 ms400 ms600 ms800 ms1000 ms1200 ms1400 ms1

Name

cran.rstudio.com

R.css

logo.html

navbar.html

banner.shtml

c3becaf1-ce23-4f76-8c40-7d1dc...

6d0496d5-10e0-4bc4-8de7-630...

588017fb-4bb3-479d-aa78-c721...

eaefc310-c845-491a-9995-5716...

5fb38acc-f76d-4940-a9c6-6db3...

695eff82-97c9-43f3-9c04-4356...

8509dbfb-bddb-4841-9f23-9709...

R.css

Rlogo.svg

× Headers Preview Response Cookies Timing

▼ General

Request URL: https://cran.rstudio.com/

Request Method: GET

Status Code: 🟢 304 Not Modified

Remote Address: 54.230.146.5:443

▼ Response Headers view source

Cache-Control: max-age=1800

Connection: keep-alive

Date: Mon, 20 Jun 2016 22:30:16 GMT

ETag: "57b14b7-352-52743447eb540"

Expires: Mon, 20 Jun 2016 23:00:16 GMT

Server: Apache/2.2.22 (Ubuntu)

Vary: Accept-Encoding

Via: 1.1 30bb04916f91d64c600e15c15000042d.cloudfront.net (CloudFront)

X-Amz-Cf-Id: SGe5Hp1yxls_9Woo9Djn6P4ALW-VwfsxCD3en9MTNuYk7gsCiUeR6A==

X-Cache: Miss from cloudfront

▼ Request Headers view source

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8

Accept-Encoding: gzip, deflate, sdch, br

Accept-Language: en-US,en;q=0.8,es;q=0.6

Cache-Control: max-age=0

Connection: keep-alive

Cookie: __utma=12992436.965097718.1451586082.1453302504.1453390119.5; __utmc=12992436; __utmz=C7B0B8; _biz_ABTestA=%5B1786%2C1662%5D; _vis_opt_exp_4_split=2; _biz_flagsA=%7B%22Version%22%3A%221.1%22%22%7D; _vwo_uuid_v2=14F12207CA1D33D2E33626E99C2767A4|8c67417811dc8dc9d3512f952922b75; _vis_opt_teoken:_mch-rstudio.com-1457506544126-48968; _biz_pendingA=%5B%5D

Host: cran.rstudio.com

Modified Since: Sat, 10 Dec 2015 17:05:40 GMT

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*

03:00

```
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

GET

Read

POST

Create

PUT

Update

DELETE

Delete

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

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

issue
#

base url

path

body

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



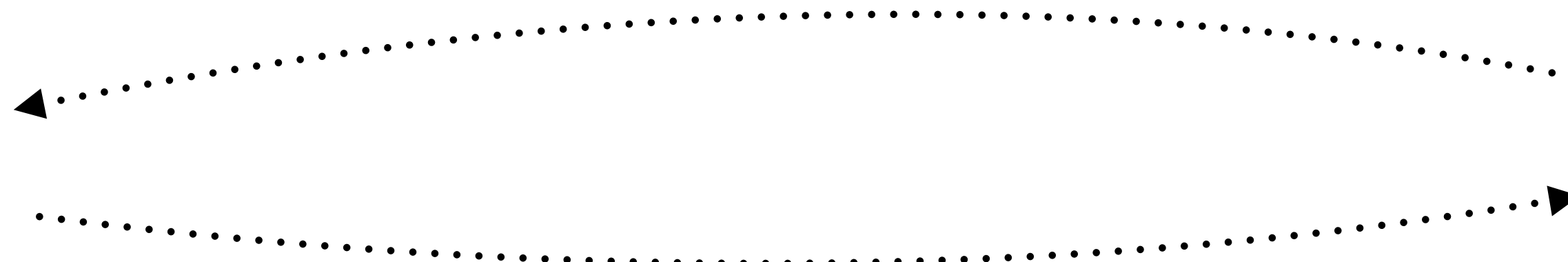
HTTP Verbs: DELETE

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

04:00

```
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>>

02:00

418: “I’m a teapot”

<https://http.cat/418>



418
I'm a teapot

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



The diagram consists of two curved arrows. The first arrow starts at the text 'raw bytes' on the right and points to the '70' character in the output of 'x\$content'. The second arrow starts at the text 'to extract data' on the right and points to the 'content(x)' function call.

to extract data

More in Part II

Your Turn

Using <http://httpbin.org/get>

- Get status code from an http response object - Use http 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()?

05:00

```
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/MV5BMTQ1MjQwMTE5OF5BMTI5BanBnXkFtZTgwNjk3MTcyMDE@._V1_SX300.jpg",
  "Metascore": "74",
  "imdbRating": "7.6",
  "imdbVotes": "410,734",
  "imdbID": "tt2294629",
  "Type": "movie",
  "Response": "True"
}
```

JSON

- Javascript **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

- Extensible **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.

04:00

```
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