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 State Transfer

an architectural style in which most web APIs are constructed

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

HyperText Transfer Protocol

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
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
#> ... 

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

GET
POST
PUT
DELETE

Read
Create
Update
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


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

```json
{
  "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
```

- 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")`

**Method**

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

**Headers**

`httr::add_headers(hello = "world")`

**Body**

`httr::POST(body = list(foo = "bar"))`
httpbin.org

httpbin(1): HTTP Request & Response Service

Freely hosted in HTTP, HTTPS & EU flavors by Runscape

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 <- GET(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 ...

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
res

# status code: 500
res <- GET("http://httpbin.org/status/500")
res
res

# the content isn’t always empty! Look in content AND headers for error messages
Data Formats
JSON

• Javascript Object Notation

• 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
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
<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 "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/MV5BMTQ1MjQwMTE5OF5BMl5BanBnXkFtZTgwNjk3MDEyMDQ2._V1_SX300.jpg" metascore="74" imdbRating="7.6" imdbVotes="410,734" imdbID="tt2294629" type="movie"/>
</root>
XML

• **Extensible Markup Language**

• 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
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")

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

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

c 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**
- **Status / Headers / Body**

HTTP **request**

HTTP **response**

Data formats: **JSON** and **XML**
thank you