APIs

INFO 201

Joel Ross

Winter 2017

https://slides.com/joelross/info201w17-apis/live

Today's Objectives

By the end of class, you should be able to

- Understand how data is provided by **web APIs**
- **Access** data from a **RESTful API**
- Work with structured data that is formatted in **JSON**

https://api.github.com/users/info201-w17/repos

Web API

Many **web services** allow you to access their data through their **web API**.

E.g., **GitHub**! (https://developer.github.com/v3/)

https://api.github.com/users/your-github-name/repos

```
# command-line downloading
curl https://api.github.com/users/info201-w17/repos

# authenticate (to see private repos)
curl -u username https://api.github.com/repos/info201-w17/repos

# include headers
curl -i https://api.github.com/users/info201-w17/repos
```

API

An **A**pplication **P**rogramming **I**nterface.

The *interface* we can use to interact with an *application* through *programming*.

Interface

The point at which two components meet and *communicate*.

An Interface

```
DoSomething <- function(arg.1, arg.2) {
  # do some work here

  return(result)  # return the result
}
```

Web API Examples

Spotify

imgur

twitter

github
SOCIAL CODING

reddit

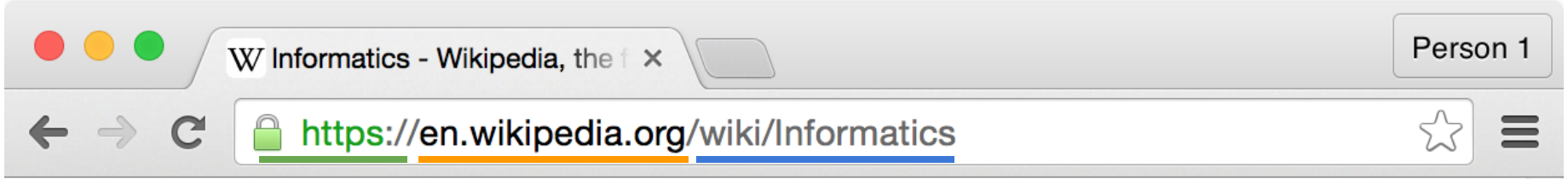
SDOT
Seattle Department of Transportation

SUNLIGHT
FOUNDATION

UNHCR
The UN Refugee Agency

http://www.programmableweb.com/

HTTP Requests

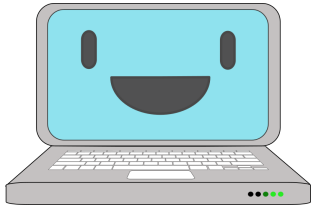


protocol (how to ask for data) | host (who has the data) | path (what data you want)

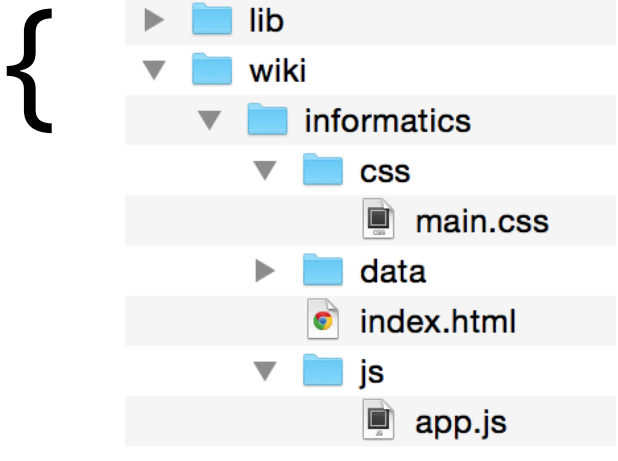
Request

"Hi Wikipedia, I'd like you to send me the wiki/Informatics data!"

Web Service



Response



URI: Uniform Resource Indicator

HTTP requests are sent to a particular **resource** on the web, identified by its **URI** (think: web address).

`http://www.domain.com/users` => a list of users

- The address is the *identifier* , the list is the *resource*
- Can have *subresources*: `http://www.domain.com/user/joel` => a specific user

URI Format

Like postal addresses, **URIs** follow a particular format.

https://www.domain.com:9999/example/info/page.html?type=husky&name=dubs#nose

scheme domain port path query fragment

- `scheme (protocol)` how to access the information
- `domain` which web service has the resource
- `path` what resource to access
- `query` extra ***parameters*** (arguments) to the request

  format: `?key=value&key=value&key=value`

API URIs

A web service's URI has two parts:

1. The **Base URI**

   `https://api.github.com`

2. An **EndPoint**

   A variable (also `:username`)

   `/users/{username}/repos`

   `/repos/:owner/:repo`

   `/search/repositories`

   `/emojis`

https://developer.github.com/v3/

HTTP Verbs

HTTP requests include a target resource and a **verb** (method) specifying what to do with it

`GET`	Return a representation of the current state of the resource
`POST`	Add a new subresource (e.g., insert a record)
`PUT`	Update the resource to have a new state
`PATCH`	Update a portion of the resource's state
`DELETE`	Remove the resource
`OPTIONS`	Return the set of methods that can be performed on the resource

HTTP Requests in R

We can send HTTP Requests (a **verb** and a **URI**) using the `httr` package.

```
# Install `httr` package
# Only needs to be done once per machine!
install.packages("httr")

# Load the package (tell R functions are available for use)
library("httr")
```

```
# GET request for iSchool home page
GET("https://ischool.uw.edu/")

# GET request to search google
query.params <- list(q = "informatics")
GET("https://www.google.com/search", query = query.params)

# GET request for GitHhub repos
base.uri <- "https://api.github.com"
resource <- paste0("/users/","info201-w17","/repos")
GET(paste0(base.uri, resource))
```

https://www.rdocumentation.org/packages/httr/versions/1.2.1

REST Architecture

REpresentational State Transfer

“ The architecture of the web

Treat all data as a **resource** with a unique identifier (URI), and use ***typed HTTP requests*** to interact with those data.

A web service is "**RESTful**" if it conforms to this pattern.

Developed by Roy Fielding, UCI Informatics!

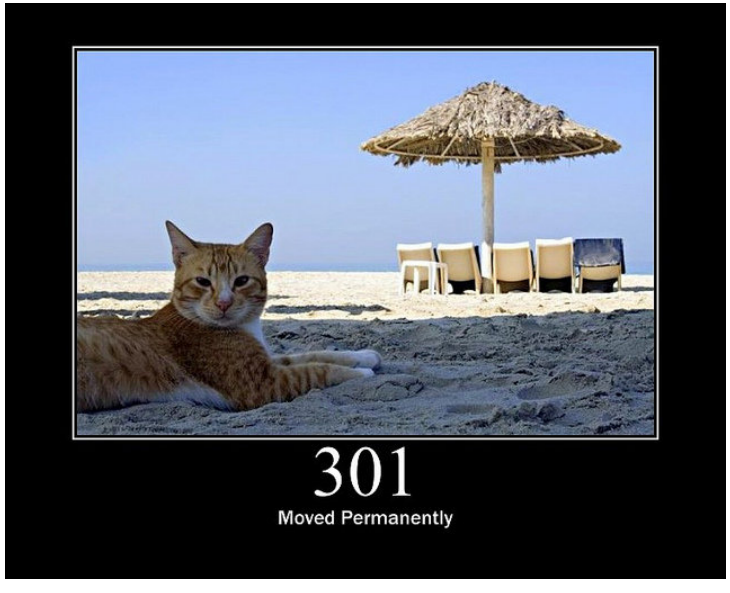
HTTP Responses

The **response** to an HTTP request (returned by the `GET()` function) has two parts:

1. **Header** with information about the response. Like a postal envelope.

2. **Body** with the ***content*** (data) of the response. Like a postal letter.

HTTP Response Codes



source
api

Accessing Content

Use the **httr::content()** function to access the content of an HTTP response.

content() variable
in **httr** package

```
# GET request for iSchool home page
response <- GET("https://ischool.uw.edu/")

# extract the body from the response
body <- content(response, "text")
```

Extract content as text (a string)

JSON

Java Script Object Notation

```
{
  "first_name": "Ada",
  "job": "Programmer",
  "salary": 78000,
  "in_union": true,
  "favorites": {
    "music": "jazz",
    "food": "pizza",
  }
}
```

JSON Objects

JSON data is structured into **key-value pairs** called **objects**. These are similar to **R lists**.

written in braces

colon between key & value

```
{
  "first_name": "Ada",
  "job": "Programmer",
  "salary": 78000,
  "in_union": true,
  "favorites": {
    "music": "jazz",
    "food": "pizza",
  }
}
```

keys as strings

```
list(
  first.name = "Ada",
  job = "Programmer",
  salary = 78000,
  in.union = TRUE,
  favorites = list(
    music = "jazz",
    food = "pizza"
   )
)
```

A nested list within the list!

JSON Arrays

JSON data can also include **arrays**, which are like **untagged lists** (not vectors because can mix types!)

written in brackets

```
[
  "Aardvark",
  "Baboon",
  "Camel",
  12,
  false,
  ["fido","spot","rover"]
]
```

```
list(
  "Aardvark",
  "Baboon",
  "Camel",
  12,
  FALSE,
  list("fido","sparky")
)
```

A nested list within the list!

Arrays of Objects

Data returned from an API very commonly contains **arrays** of **objects**:

These can be converted into **data frames**!

row

```
[
  { "opponent": "Dolphins", "sea_score": 12, "opp_score": 10 },
  { "opponent": "Rams", "sea_score": 3, "opp_score": 9 },
  { "opponent": "49ers", "sea_score": 37, "opp_score": 18 },
  { "opponent": "Jets", "sea_score": 27, "opp_score": 17 },
  { "opponent": "Falcons", "sea_score": 26, "opp_score": 24 }
]
```

column name

column value

Parsing JSON

We can convert **JSON Strings** into **R** data structures (e.g., lists, data frames) using the `jsonlite` package.

```
# Install `jsonlite` package
# Only needs to be done once per machine!
install.packages("jsonlite")

# Load the package (tell R functions are available for use)
library("jsonlite")
```

```
# an example string of JSON
json <- '{"first_name":"Ada","job":"Programmer","pets":["rover","flu

# convert from string into R structure
ada <- fromJSON(json)  # a list!


response <- GET("https://api.github.com/users/info201-w17/repos")
body <- content(response, "text")
repos <- fromJSON(body)  # a data frame!
```

https://www.rdocumentation.org/packages/jsonlite/versions/1.2

Inspecting JSON

APIs will not always provide JSON that easily converts to a data frame. You will need to **inspect** the returned data to find what piece of it you want to work with!

```
# a GitHub search for `dplyr`
uri <- "https://api.github.com/search/repositories?q=dplyr"
response <- GET(uri)
body.data <- fromJSON(content(response, "text"))  # extract and parse

# is it a data frame already?
is.data.frame(body.data)  # FALSE

# inspect the data!
str(body.data)  # view as a formatted string
names(body.data)  # view the tag names
  # looking at the JSON data itself (e.g., in the browser),
  # `items` is the key that contains the value we want

# extract the (useful) data
items <- body.data$items  # extract from the list
is.data.frame(items)  # TRUE; we can work with that!
```

Nested Data Frames

Because JSON encourages **nested lists** (lists within lists), parsing a JSON string will likely produce a data frame whose columns are themselves data frames!

```
# Let's do something silly
people <- data.frame(names = c('Spencer', 'Jessica', 'Keagan'))

favorites <- data.frame(  # a data frame with two columns
                food = c('Pizza', 'Pasta', 'salad'),
                music = c('Bluegrass', 'Indie', 'Electronic')
            )
# Store dataframe column
people$favorites <- favorites  # make `favorites` column a data frame!

# this prints nicely...
print(people)

# but doesn't actually work like we expect!
people$favorites.food  # NULL
people$favorites$food  # [1] Pizza Pasta salad
```

Flattening JSON

Use the **jsonlite::flatten()** function to take the columns of the nested data frame and turn them into columns in the "outer" frame.

```
# Previous example...
people <- data.frame(names = c('Spencer', 'Jessica', 'Keagan'))
favorites <- data.frame(  # a data frame with two columns
                food = c('Pizza', 'Pasta', 'salad'),
                music = c('Bluegrass', 'Indie', 'Electronic')
             )
people$favorites <- favorites


# flatten the data frame
people <- flatten(people)

people$favorites.food  # this works as expected!
```

Accessing Web Data

In order to access and use data from a **web api**, you will need to:

1. Use `GET()` to download the data, specifying the **URI** (and any **query parameters**)
2. Use `content()` to extract the data as a JSON string (as `"text"`!)
3. Use `fromJSON()` to convert the JSON string into a list
4. Find which element in that list is your data frame of interest. You may need to go "multiple levels" in
5. Use `flatten()` to flatten that data frame
6. Analyze your data (e.g., with `dplyr`)

Module 11 exercise-1

Module 11 exercise-2

Remember to **FORK** and clone!

Will be submitting for participation

Action Items!

- Be comfortable with **module 11**
- Assignment 5 due ***Tuesday before class***

Thursday: Making reports with *R Markdown*