Web Development and API Design

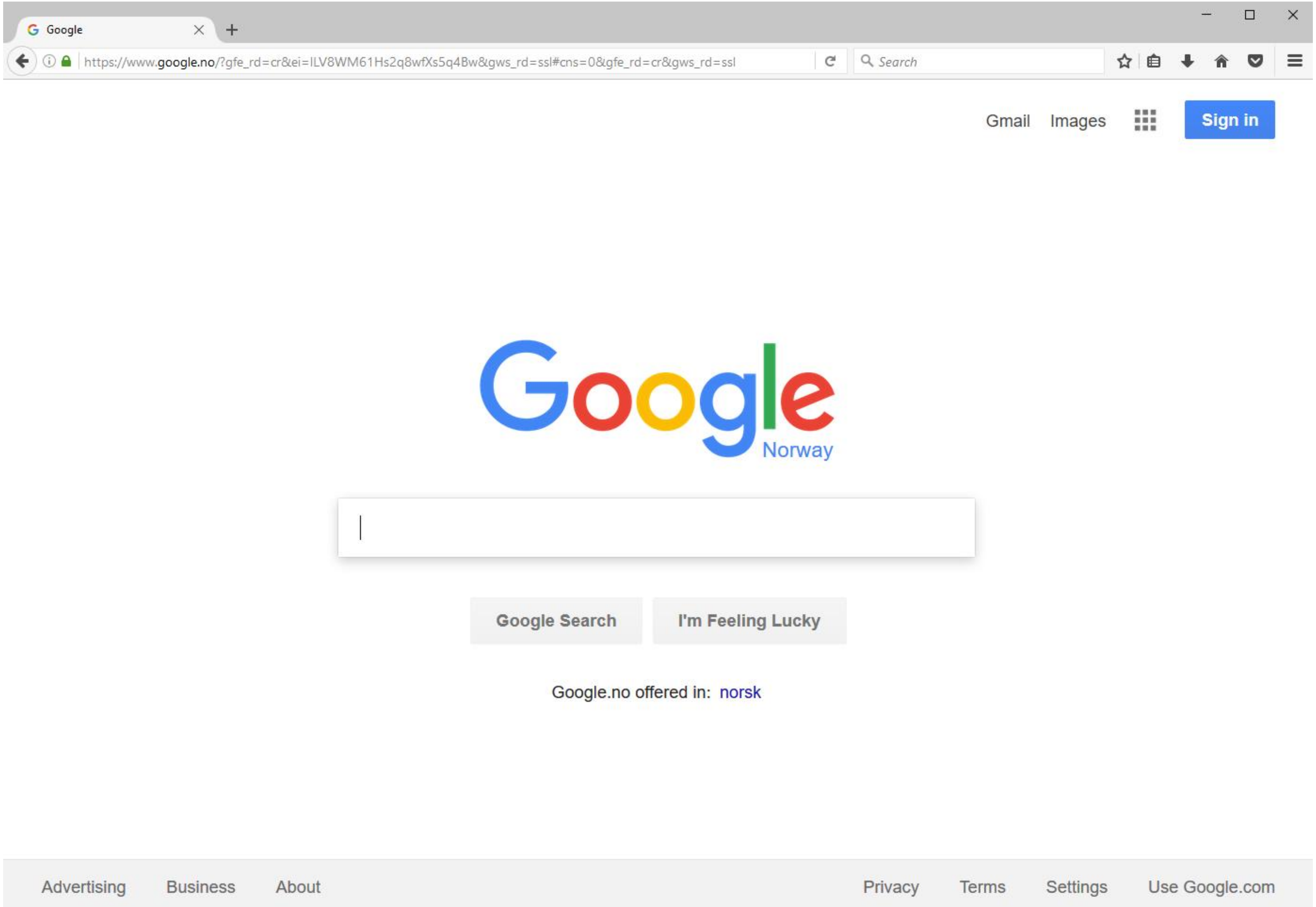
Lesson 07: RESTful APIs Theory

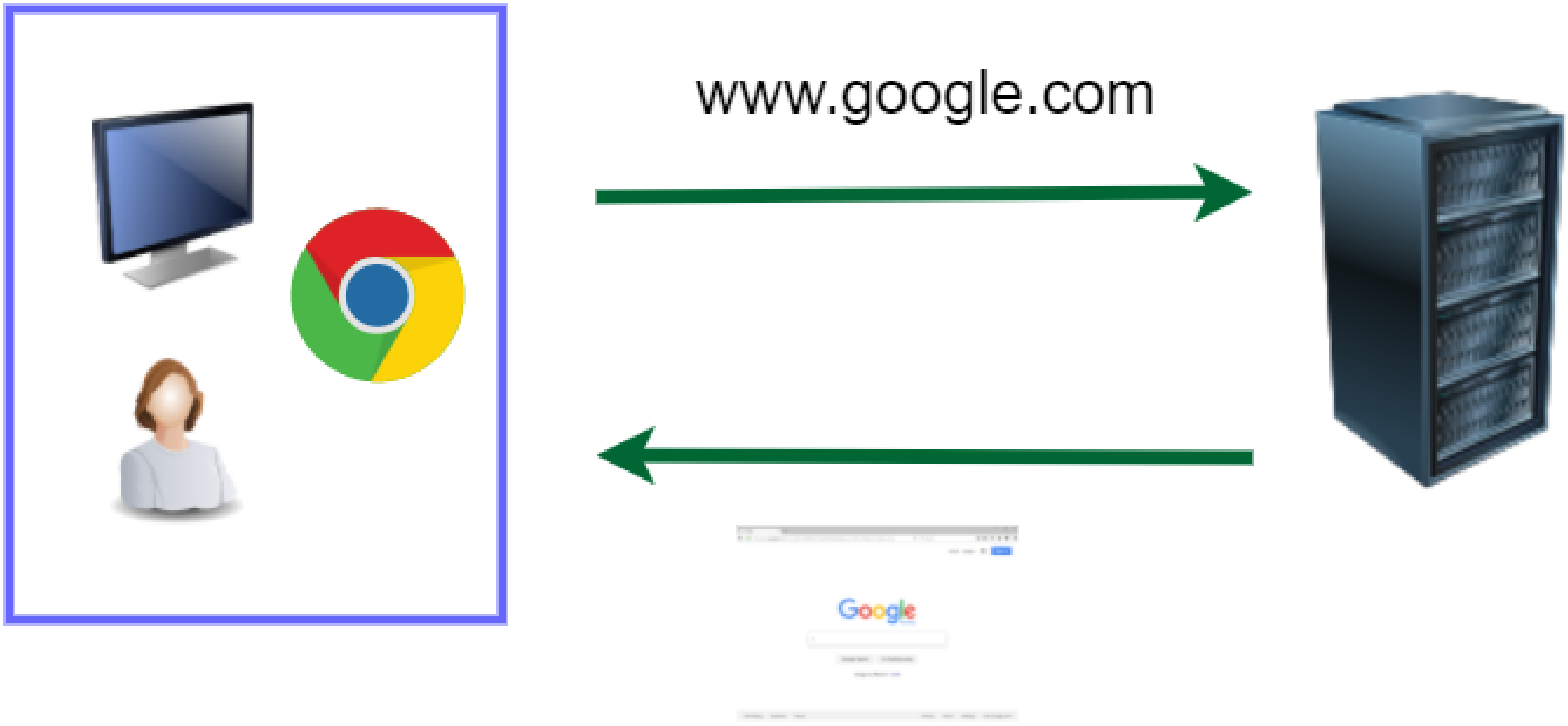
Prof. Andrea Arcuri

Goals

- Revision of URLs and HTTP
- Understand the main concepts of REST web services
- Understand differences and similarities between handling of static resources and dynamic content
    - e.g., HTML/CSS files vs. JSON data

HTTP

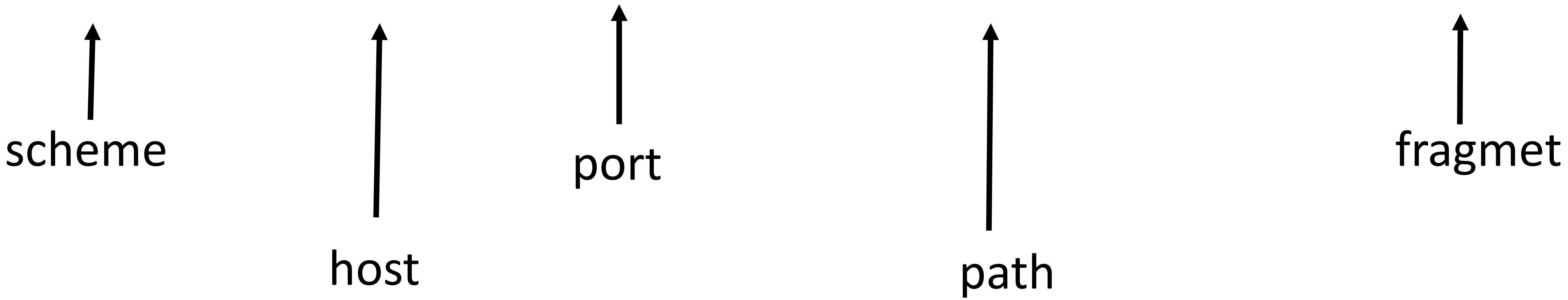

Google
https://www.google.no/?gfe_rd=cr&ei=ILV8WM61Hs2q8wfXs5q4Bw&gws_rd=ssl#cns=0&gfe_rd=cr&gws_rd=ssl
Search
Gmail
Images
Sign in
Google
Norway
Google Search
I'm Feeling Lucky
Google.no offered in: norsk
Advertising
Business
About
Privacy
Terms
Settings
Use Google.com



Send a HTTP request over TCP, and get back a HTML page which will be visualized in the browser

URL (Uniform Resource Locator)

- Reference to a web resource and how to retrieve it
- **scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]**
- https://en.wikipedia.org:443/wiki/Uniform_Resource_Locator#Syntax



scheme

host

port

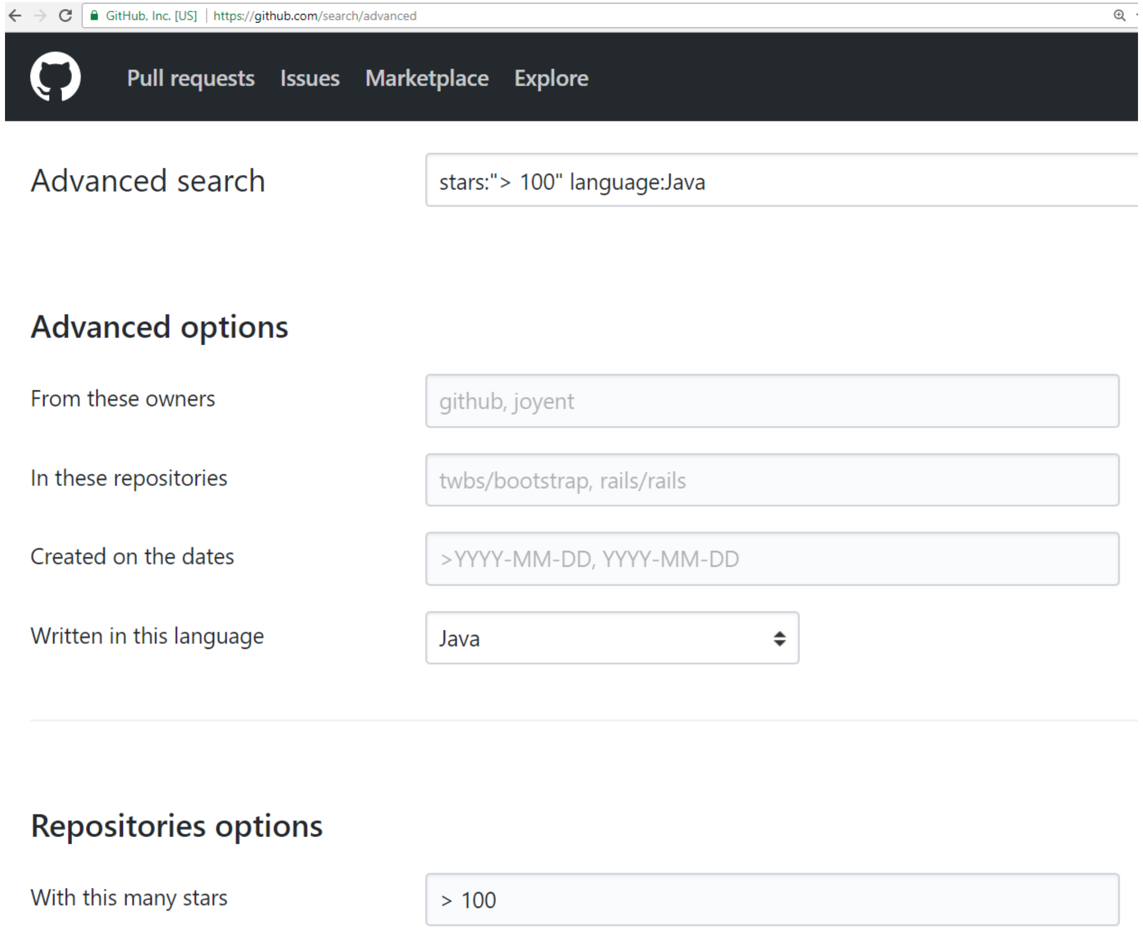
path

fragmet

Cont.

- **Scheme**: how to access the resource
  - http, https, file, ftp, etc.
- **Host**: the name of the server, or directly its numeric IP address
- **Port**: the listening port you will connect to on the remote server
- **Path**: identifies the resource, usually in a hierarchical format
  - Eg, /a/b/c
- **Query**: starting with “?”, list of <key>=<value> properties, separated by “&”
  - eg https://github.com/search**?q=java&type=Repositories&ref=searchresults**
- **Fragment**: identifier of further resource, usually inside the main you requested
  - Eg, a section inside an HTML page

- https://github.com/search?utf8=%E2%9C%93&q=stars%3A%22%3E+100%22+language%3AJava&type=Repositories&ref=advsearch&l=Java&l=
- The asked page/resource is **/search**, where it is retrieved in different ways based on the list of query "?" parameters



GitHub, Inc. [US] | https://github.com/search/advanced

Pull requests Issues Marketplace Explore

Advanced search: stars:"> 100" language:Java

Advanced options

From these owners: github, joyent

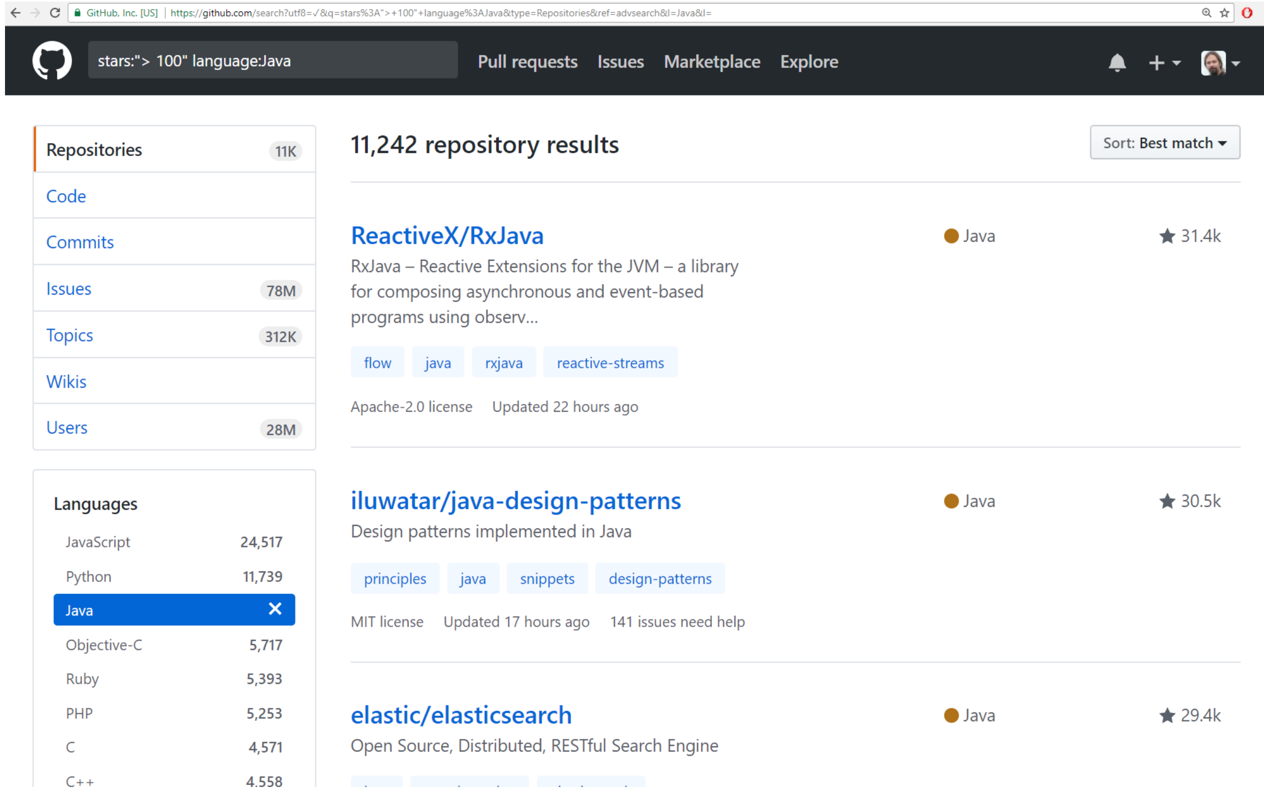
In these repositories: twbs/bootstrap, rails/rails

Created on the dates: >YYYY-MM-DD, YYYY-MM-DD

Written in this language: Java

Repositories options

With this many stars: > 100



GitHub, Inc. [US] | https://github.com/search?utf8=✓&q=stars%3A">+100"+language%3AJava&type=Repositories&ref=advsearch&l=Java&l=

stars:"> 100" language:Java | Pull requests Issues Marketplace Explore

Repositories	11K
Code	
Commits	
Issues	78M
Topics	312K
Wikis	
Users	28M

Languages

JavaScript	24,517
Python	11,739
Java	✕
Objective-C	5,717
Ruby	5,393
PHP	5,253
C	4,571
C++	4,558

11,242 repository results

Sort: Best match ▾

ReactiveX/RxJava — ● Java — ★ 31.4k
RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observ...
flow java rxjava reactive-streams
Apache-2.0 license Updated 22 hours ago

iluwatar/java-design-patterns — ● Java — ★ 30.5k
Design patterns implemented in Java
principles java snippets design-patterns
MIT license Updated 17 hours ago 141 issues need help

elastic/elasticsearch — ● Java — ★ 29.4k
Open Source, Distributed, RESTful Search Engine

URI (Uniform Resource Identifier)

- String of characters used to identify a resource
- A URL is a URI:
  - Exactly same format
  - In URL, the resource is typically located on a network
  - Given a URL, you should be able to access the resource, which is not necessarily true for URI
- The distinction between URL and URI is conceptually very thin
  - Most people use the two terms interchangeably

TCP not Enough

https://en.wikipedia.org:443/wiki/Uniform_Resource_Locator

↑

host:port

- Host and port are needed to establish a TCP connection
- But what data should we send to specify that we want to retrieve the HTML page at that location?

HTTP History

- Protocol Used to specify structure of messages
- Started at CERN in 1989
- 1995: version 0.9
- 1996: version 1.1
- 1999: “updates” to 1.1
- 2014: more “updates” to 1.1
- 2015: version 2.0

Http Versioning: What a Mess!!!

- HTTP is one the **worst** examples of versioning done **wrong**
- Changing specs and semantics over 18 years, but still keeping the same version number **1.1**!!!
- Why? To support the largest number of browsers, even very old ones
- Not many people realized there was an update in 2014… you might still find quite a few libraries/tools that wrongly use the 1999 version

RFC (Request for Comments)

Technically, a RFC is not a “standard” yet, but it is de-facto in practice

- RFC 7230, HTTP/1.1: Message Syntax and Routing
- RFC 7231, HTTP/1.1: Semantics and Content
- RFC 7232, HTTP/1.1: Conditional Requests
- RFC 7233, HTTP/1.1: Range Requests
- RFC 7234, HTTP/1.1: Caching
- RFC 7235, HTTP/1.1: Authentication
- RFC 7540, HTTP/2
- Etc.
- *When working with web services, it is fundamental to understand all the low level details of HTTP*

HTTP 1.1 vs 2

- v2 is quite recent (2015), and still not so common
- Unless otherwise stated, we will just deal with v1.1
- From user’s perspective, v2 is like v1.1
  - Same methods/verbs, just better optimization / performance improvement
  - More like adding functionalities, not replacing it
- Main visible difference: v1.1 is “text” based, whereas v2 has its own byte format (less space, but more difficult to read/parse for humans)

HTTP Messages: 3 Main Parts

- First line specifying the action you want to do, eg GET a specific resource
- Set of *headers* to provide extra meta-info
  - eg in which format you want the response: JSON? Plain Text? XML?
  - In which language? Norwegian? English?
- (Optional) Body: can be anything.
  - Request: usually to provide user data, eg, login/password in a submitted form
  - Response: the actual resource that is retrieved, eg a HTML page

First line

- <METOD> <RESOURCE> <PROTOCOL> \r\n
- Ex.: GET / HTTP/1.1
  - <method> **GET**
  - <resource> **/**
  - <protocol> **HTTP/1.1**
- A resource can be anything
  - html, jpeg, json, xml, pdf, etc.
- A resource is identified by its *path*
  - Recall URI, and such path is same as file-system on Mac/Linux, where "/" is the root

Different kinds of Methods

- **GET**: to retrieve a resource
- **POST**: to send data (in the HTTP body), and/or create a resource
- **PUT**: to replace an existing resource with a new one
- **PATCH**: to do a partial update on an existing resource
- **DELETE**: to delete a resource
- **HEAD**: like a GET, but only return headers, not the resource data
- OPTIONS: to check what methods are available on a resource
- TRACE: for debugging
- CONNECT: tunneling connection through proxy

Method Semantics

- Each of the methods has a clear semantics
  - Eg GET does retrieve a resource, whereas DELETE should delete it
- But *how* the application server does handle them is completely up to it
  - Eg, an application server could delete a resource when a GET is executed

Verbs should not be in paths

- Given a resource “/x.html”
- **Wrong**: GET on “www.foo.org/x.html/delete” to delete “x.html”
  - Here the resource would be “/x.html/delete”
- Also wrong to use query, eg “www.foo.org/x.html?*method=delete*”
- Paths should represent/identify resources, and NOT actions on those

Idempotent Methods

RFC 7231: “A request method is considered *idempotent* if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request...

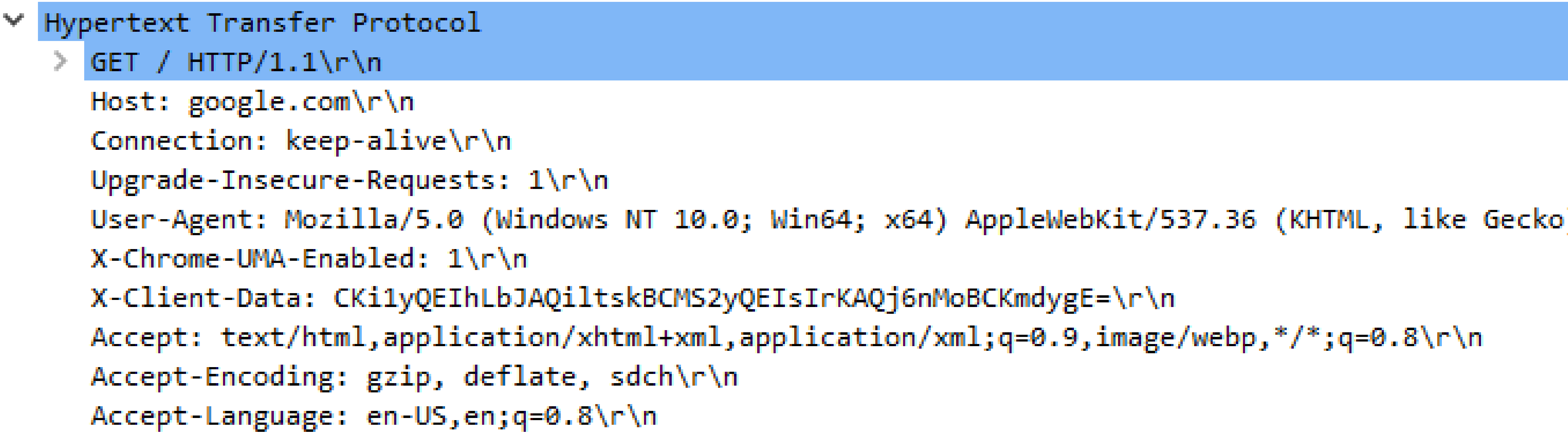
… if a client sends a … request and the underlying connection is closed before any response is received, then the client can establish a new connection and retry the idempotent request.”

Which methods are idempotent?

GET	✓
POST	✗
DELETE	✓
PUT	✓
PATCH	✗
HEAD	✓

Headers

- Extra meta-information, besides Method/Resource
- Pairs <key>:<value>
- For example:
  - In which format am I expecting the resource? HTML? JSON?
  - In which language do I want it?
  - Who am I? (important for user authentication)
  - Should the TCP connection be kept alive, or should it be closed after this HTTP request?
  - Etc.



```
Hypertext Transfer Protocol
  GET / HTTP/1.1\r\n
  Host: google.com\r\n
  Connection: keep-alive\r\n
  Upgrade-Insecure-Requests: 1\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko
  X-Chrome-UMA-Enabled: 1\r\n
  X-Client-Data: CKi1yQEIhLbJAQiltskBCMS2yQEIsIrKAQj6nMoBCKmdygE=\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
  Accept-Encoding: gzip, deflate, sdch\r\n
  Accept-Language: en-US,en;q=0.8\r\n
```

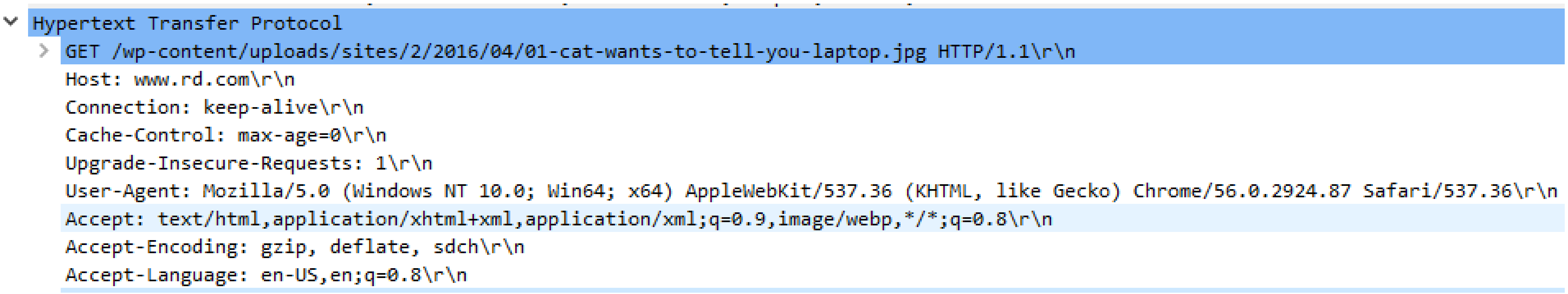
- Request for *www.google.com* in a browser (eg Chrome)
- Recall, you can use WireShark or Chrome Developer Tools
- Several HTTP headers: eg, including preferred format and language

HTTP Body

- After last header, there must be an empty line
- Any data after that, if any, would be part of the payload, ie HTTP body
- Request: needed for **POST**, **PUT** and **PATCH**
- Response: needed for **GET** (also the other methods “might” have body, but **HEAD**)

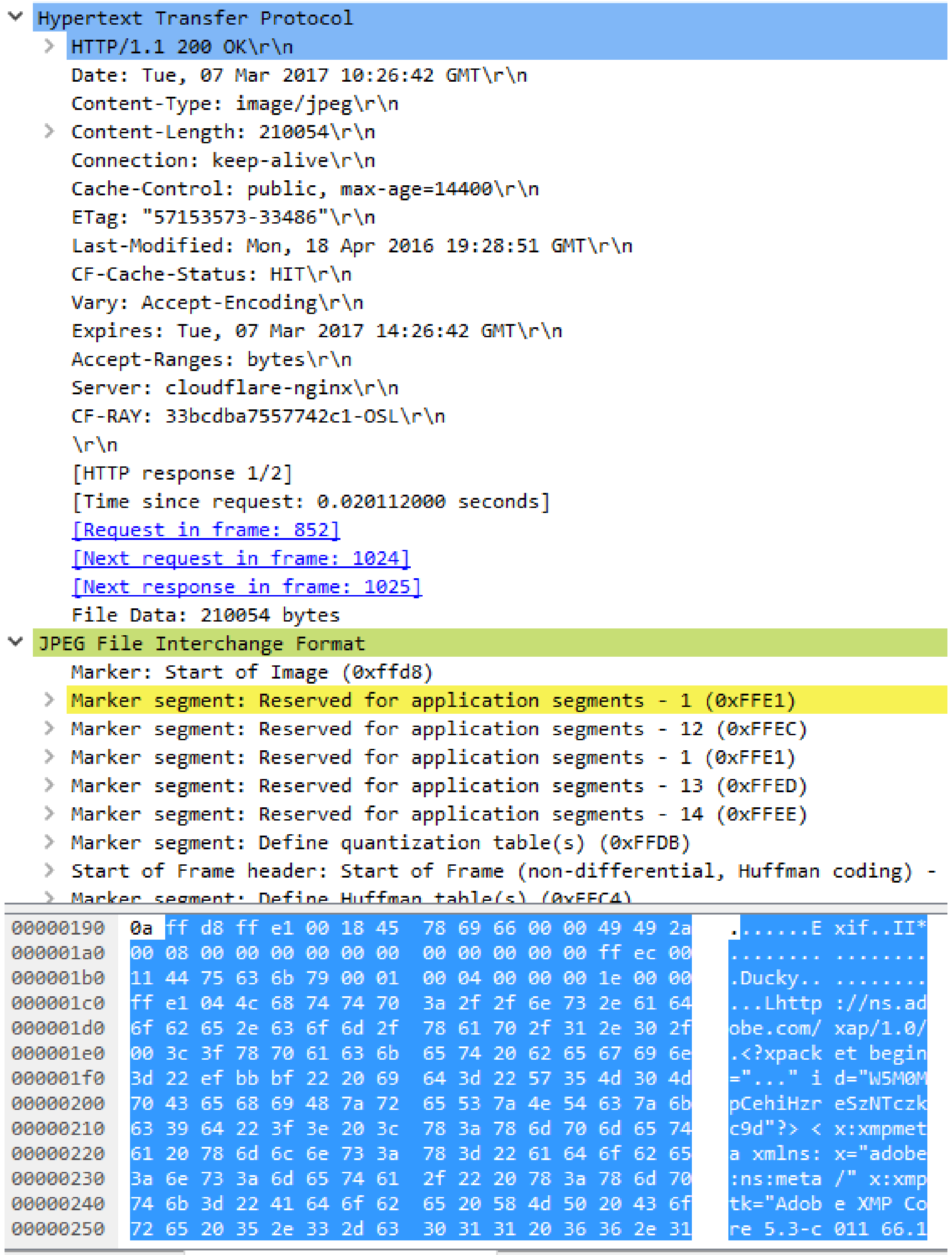
http://*www.rd.com***/wp-content/uploads/sites/2/2016/04/01-cat-wants-to-tell-you-laptop.jpg**

GET with no body



```
Hypertext Transfer Protocol
  GET /wp-content/uploads/sites/2/2016/04/01-cat-wants-to-tell-you-laptop.jpg HTTP/1.1\r\n
  Host: www.rd.com\r\n
  Connection: keep-alive\r\n
  Cache-Control: max-age=0\r\n
  Upgrade-Insecure-Requests: 1\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
  Accept-Encoding: gzip, deflate, sdch\r\n
  Accept-Language: en-US,en;q=0.8\r\n
```

Have a look at the “Accept” header… there is no “jpg” there, why?



```
v Hypertext Transfer Protocol
  > HTTP/1.1 200 OK\r\n
    Date: Tue, 07 Mar 2017 10:26:42 GMT\r\n
    Content-Type: image/jpeg\r\n
  > Content-Length: 210054\r\n
    Connection: keep-alive\r\n
    Cache-Control: public, max-age=14400\r\n
    ETag: "57153573-33486"\r\n
    Last-Modified: Mon, 18 Apr 2016 19:28:51 GMT\r\n
    CF-Cache-Status: HIT\r\n
    Vary: Accept-Encoding\r\n
    Expires: Tue, 07 Mar 2017 14:26:42 GMT\r\n
    Accept-Ranges: bytes\r\n
    Server: cloudflare-nginx\r\n
    CF-RAY: 33bcdba7557742c1-OSL\r\n
    \r\n
    [HTTP response 1/2]
    [Time since request: 0.020112000 seconds]
    [Request in frame: 852]
    [Next request in frame: 1024]
    [Next response in frame: 1025]
    File Data: 210054 bytes
v JPEG File Interchange Format
    Marker: Start of Image (0xffd8)
  > Marker segment: Reserved for application segments - 1 (0xFFE1)
  > Marker segment: Reserved for application segments - 12 (0xFFEC)
  > Marker segment: Reserved for application segments - 1 (0xFFE1)
  > Marker segment: Reserved for application segments - 13 (0xFFED)
  > Marker segment: Reserved for application segments - 14 (0xFFEE)
  > Marker segment: Define quantization table(s) (0xFFDB)
  > Start of Frame header: Start of Frame (non-differential, Huffman coding) -
  > Marker segment: Define Huffman table(s) (0xFFC4)

00000190  0a ff d8 ff e1 00 18 45  78 69 66 00 00 49 49 2a   .......E xif..II*
000001a0  00 08 00 00 00 00 00 00  00 00 00 00 00 ff ec 00   ........ ........
000001b0  11 44 75 63 6b 79 00 01  00 04 00 00 00 1e 00 00   .Ducky.. ........
000001c0  ff e1 04 4c 68 74 74 70  3a 2f 2f 6e 73 2e 61 64   ...Lhttp ://ns.ad
000001d0  6f 62 65 2e 63 6f 6d 2f  78 61 70 2f 31 2e 30 2f   obe.com/ xap/1.0/
000001e0  00 3c 3f 78 70 61 63 6b  65 74 20 62 65 67 69 6e   .<?xpack et begin
000001f0  3d 22 ef bb bf 22 20 69  64 3d 22 57 35 4d 30 4d   ="..." i d="W5M0M
00000200  70 43 65 68 69 48 7a 72  65 53 7a 4e 54 63 7a 6b   pCehiHzr eSzNTczk
00000210  63 39 64 22 3f 3e 20 3c  78 3a 78 6d 70 6d 65 74   c9d"?> < x:xmpmet
00000220  61 20 78 6d 6c 6e 73 3a  78 3d 22 61 64 6f 62 65   a xmlns: x="adobe
00000230  3a 6e 73 3a 6d 65 74 61  2f 22 20 78 3a 78 6d 70   :ns:meta /" x:xmp
00000240  74 6b 3d 22 41 64 6f 62  65 20 58 4d 50 20 43 6f   tk="Adob e XMP Co
00000250  72 65 20 35 2e 33 2d 63  30 31 31 20 36 36 2e 31   re 5.3-c 011 66.1
```

- In this case, payload is in JPEG format
- "*Content-type*" header:
  - need to specify the format, eg JPEG. Note this is necessary because what requested by user ("Accept") might be a list, and also server might return something different
- "*Content-length*" header:
  - Essential, otherwise HTTP parser cannot know when payload is finished
- Cache handling: headers like "*Cache-Control*", "ETag", "Last-Modified", etc.
  - If visiting page for second time, no need to re-download image if hasn't changed

HTTP Response

- Same kind of headers and body as HTTP request
- Only first line does differ
- <PROTOCOL> <STATUS> <DESCRIPTION>
  - Eg, “HTTP/1.1   200   OK”
    - Note: only 1 space “ ” between the tags, I added extras just for readability
- When making a request, a lot of things could happen on server, and the “*status*” is used to say what happened

HTTP Status Codes

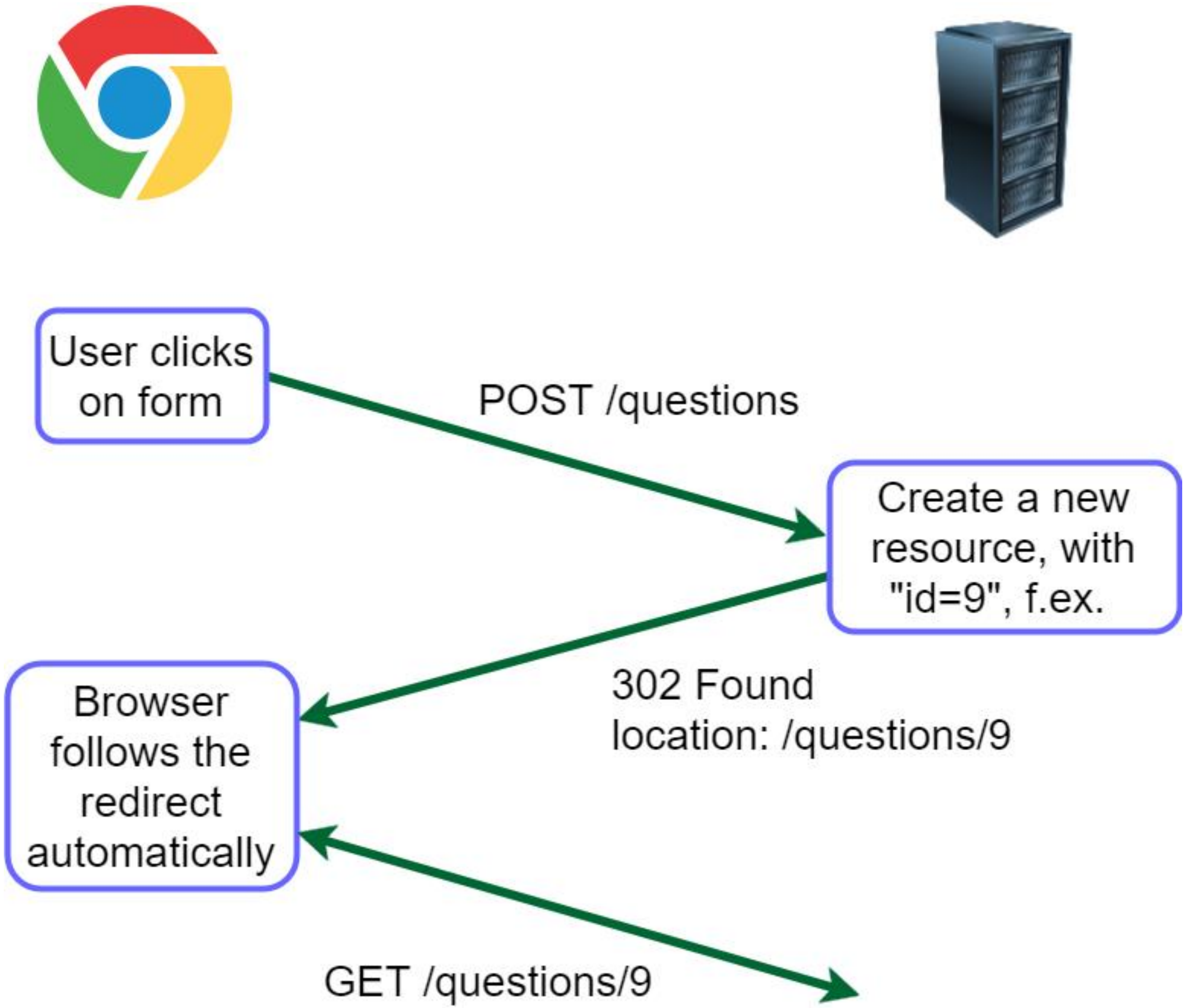
- 3 digit number, divided into “families”
- **1xx**: informational, interim response
- **2xx**: success
- **3xx**: redirection
- **4xx**: user error
- **5xx**: server error

2xx Success

- **200**: OK
- **201**: resource created
- **202**: accepted, but not completed (eg, background operation)
- **204**: no content (eg, as result of PUT or DELETE)

3xx Redirection

- Note: the semantics of these codes is a “**mess**”, being changing with different “updates” of HTTP 1.1… and being left in an inconsistent state
- **301** permanent redirection
  - If X redirects to Y, then client will never ask for X again, and go straight for Y
- **302** temporary redirection
  - “May” change verb, eg from POST to GET
- **307** temporary redirection, but same verb
  - Eg, a POST stays a POST
- “*Location*” header: URI of where we should redirect



User clicks on form

POST /questions

Create a new resource, with "id=9", f.ex.

302 Found
location: /questions/9

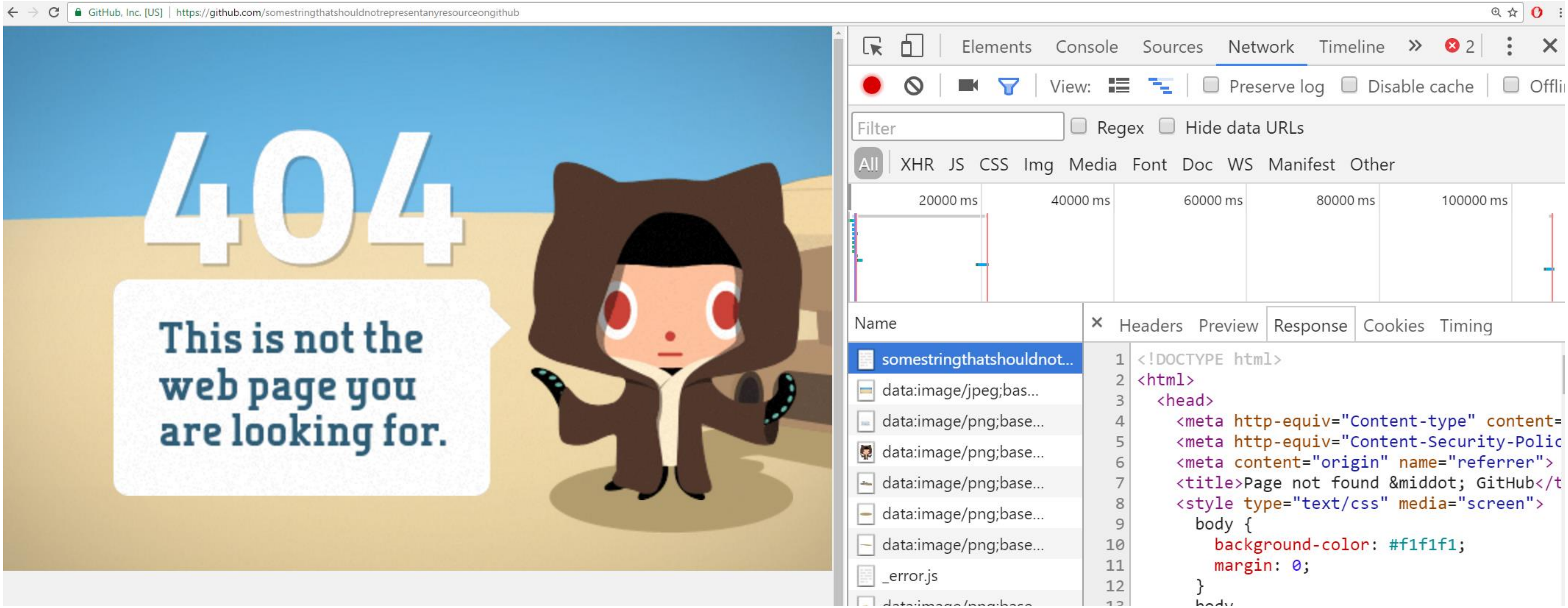
Browser follows the redirect automatically

GET /questions/9

4xx User Error

- **400**: bad request (generic error code)
- **401**: unauthorized (user not authenticated)
- **403**: forbidden (authenticated but lacking authorization, or not accessible regardless of auth)
  - Note: RFC 7231/7235 are rather ambiguous/confusing when it comes to define authentication/authorization, and differences between 401 and 403
- **404**: not found (likely the most famous HTTP status code)
- **405**: method not allowed (eg doing DELETE on a read-only resource)
- **415**: unsupported media type (eg sending XML to JSON-only server)

Even if error (eg 404), response can have a body, eg an HTML page to display



https://github.com/somestringthatshouldnotrepresentanyresourceongithub

somestringthatshouldnot...
data:image/jpeg;bas...
data:image/png;base...
data:image/png;base...
data:image/png;base...
data:image/png;base...
data:image/png;base...
_error.js
data:image/png;base...

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-type" content=
    <meta http-equiv="Content-Security-Polic
    <meta content="origin" name="referrer">
    <title>Page not found &middot; GitHub</t
    <style type="text/css" media="screen">
      body {
        background-color: #f1f1f1;
        margin: 0;
      }
      body
```

5xx Server Error

- 500: Internal Server Error
  - Often, a bug, eg an exception in the business logic (like a NPE), that propagates to the application server will be handled with a 500 response
    - Note: if whole application server does crash, then you get no response…
  - Required external services have problems, eg database connection failed
- 503: Service Unavailable
  - Eg, overloaded of requests, or scheduled downtime for maintenance

HTTPS (HTTP Secure)

- Encrypted version of HTTP, using Transport Layer Security (TLS)
- Usually on port 443 instead of 80
- URIs are the same as in HTTP (just the “scheme” does change)
- Note, the whole HTTP messages are encrypted, **but still using TCP**
  - this means it is still possible to find out the IP address and port of remote server, although cannot decipher the actual sent messages
  - this issue can be avoided by going through proxy networks like TOR , or any VPN provider (but this latter would know what you visit)

Web Services

Data/Operations Over Network

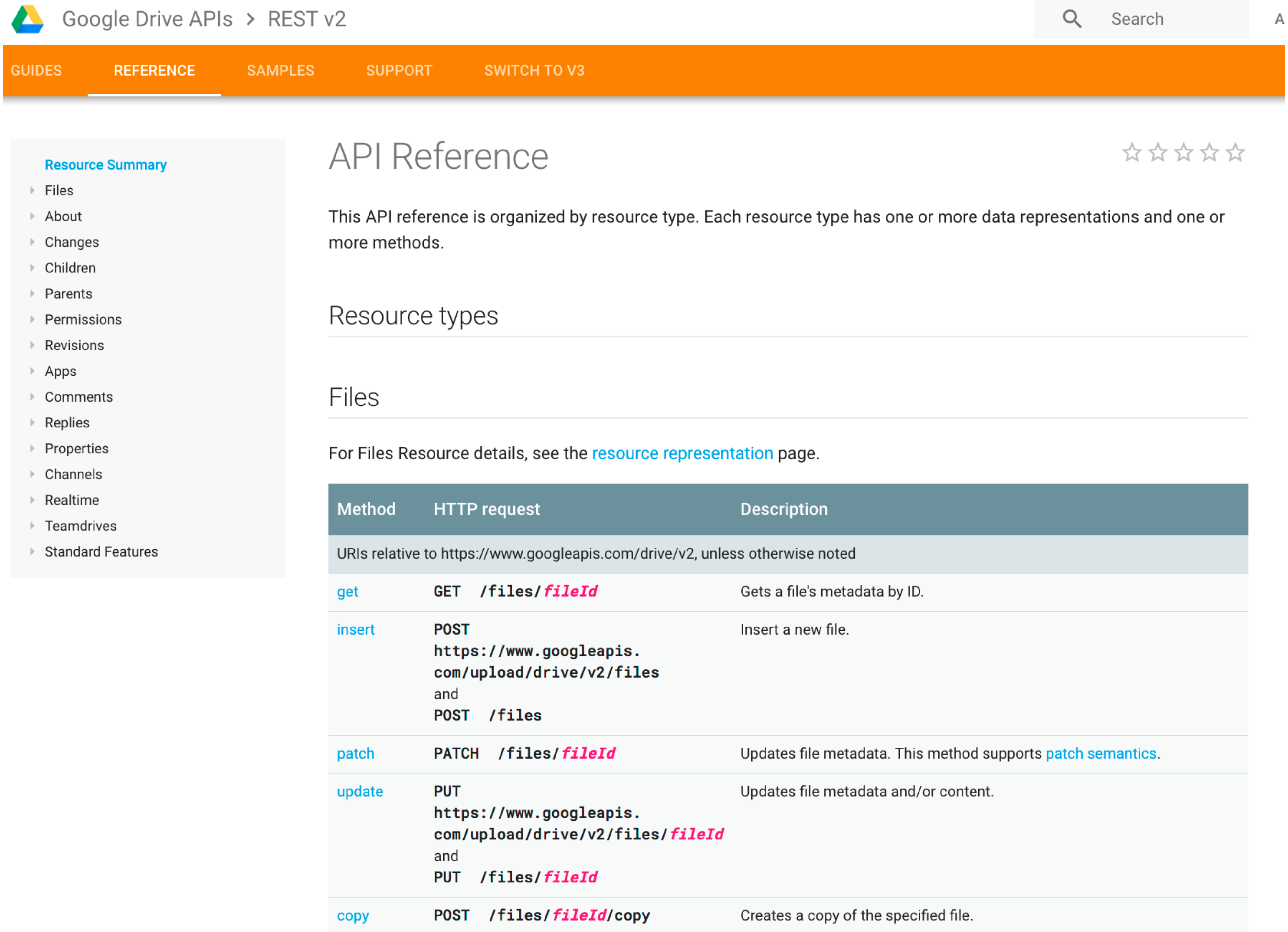
- Provide APIs over network
- Typically TCP connections
- HTTP most common protocol
- So, can see a Web Service as a process that opens a TCP port and responds to incoming requests

Types of Web Services

- REST
  - most common nowadays
  - usually strongly tied to HTTP protocol
  - *not a protocol, but set of architectural guidelines*
  - typically serving data in JSON
- SOAP
  - very common in the past, but disappearing nowadays
  - actual protocol, usually over HTTP
  - tied to XML
- GraphQL
  - the new kid

Why?

- When you want to provide programmable functionalities to your clients over the network
  - eg, see public list at http://www.programmableweb.com/
- Separation of *frontend* from *backend*
  - JavaScript doing client-side HTML rendering on browser, where backend is just a web service providing data
- *Microservice* Architecture
  - large systems split into several web services of more manageable size
  - extremely important for modern enterprise systems



Google Drive APIs > REST v2

Search

GUIDES REFERENCE SAMPLES SUPPORT SWITCH TO V3

- Resource Summary
- Files
- About
- Changes
- Children
- Parents
- Permissions
- Revisions
- Apps
- Comments
- Replies
- Properties
- Channels
- Realtime
- Teamdrives
- Standard Features

API Reference

☆☆☆☆☆

This API reference is organized by resource type. Each resource type has one or more data representations and one or more methods.

Resource types

Files

For Files Resource details, see the resource representation page.

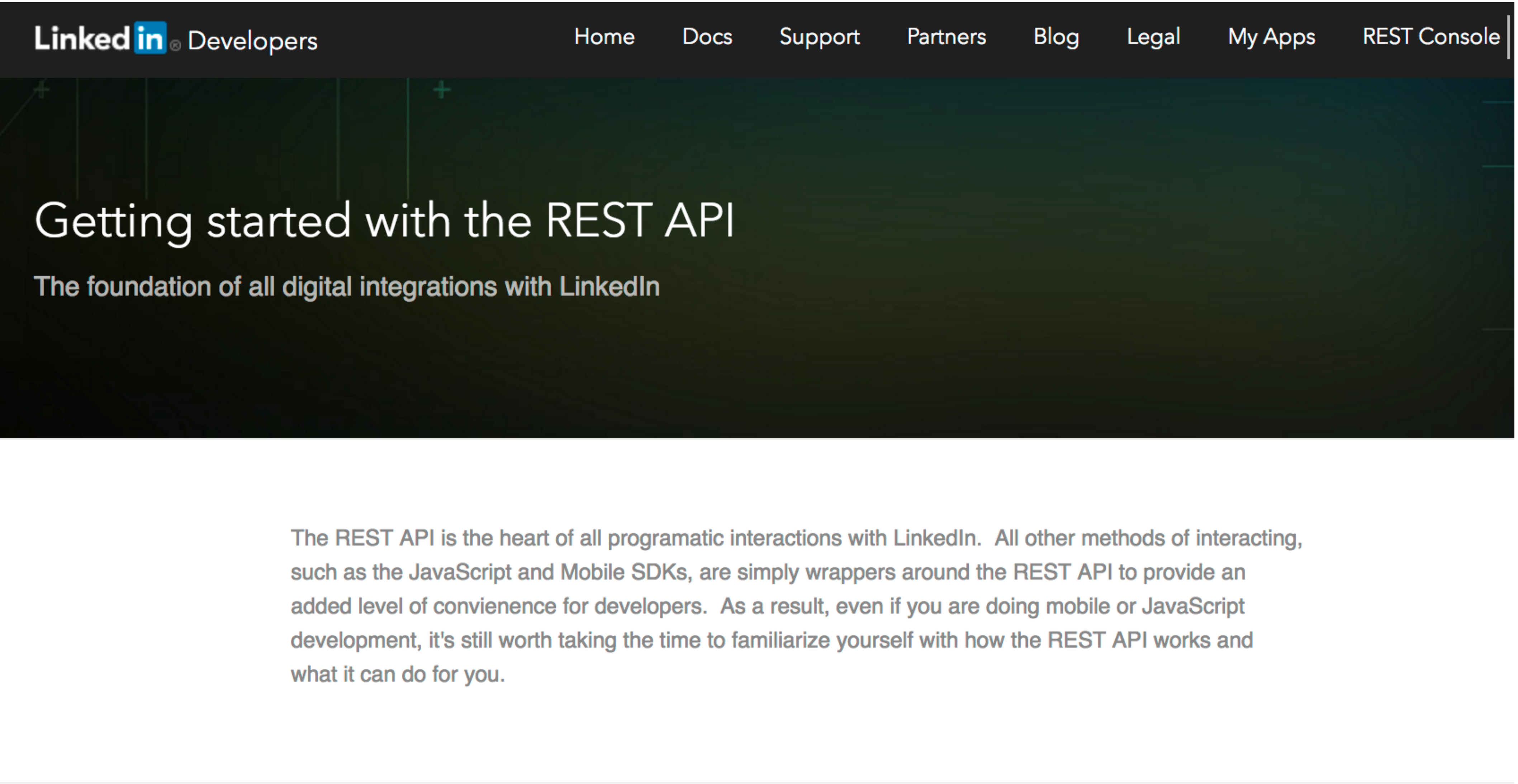
Method	HTTP request	Description
URIs relative to https://www.googleapis.com/drive/v2, unless otherwise noted		
get	`GET /files/fileId`	Gets a file's metadata by ID.

insert	`POST https://www.googleapis.com/upload/drive/v2/files`
	and
	`POST /files`
		Insert a new file.

patch	`PATCH /files/fileId`	Updates file metadata. This method supports patch semantics.

update	`PUT https://www.googleapis.com/upload/drive/v2/files/fileId`
	and
	`PUT /files/fileId`
		Updates file metadata and/or content.

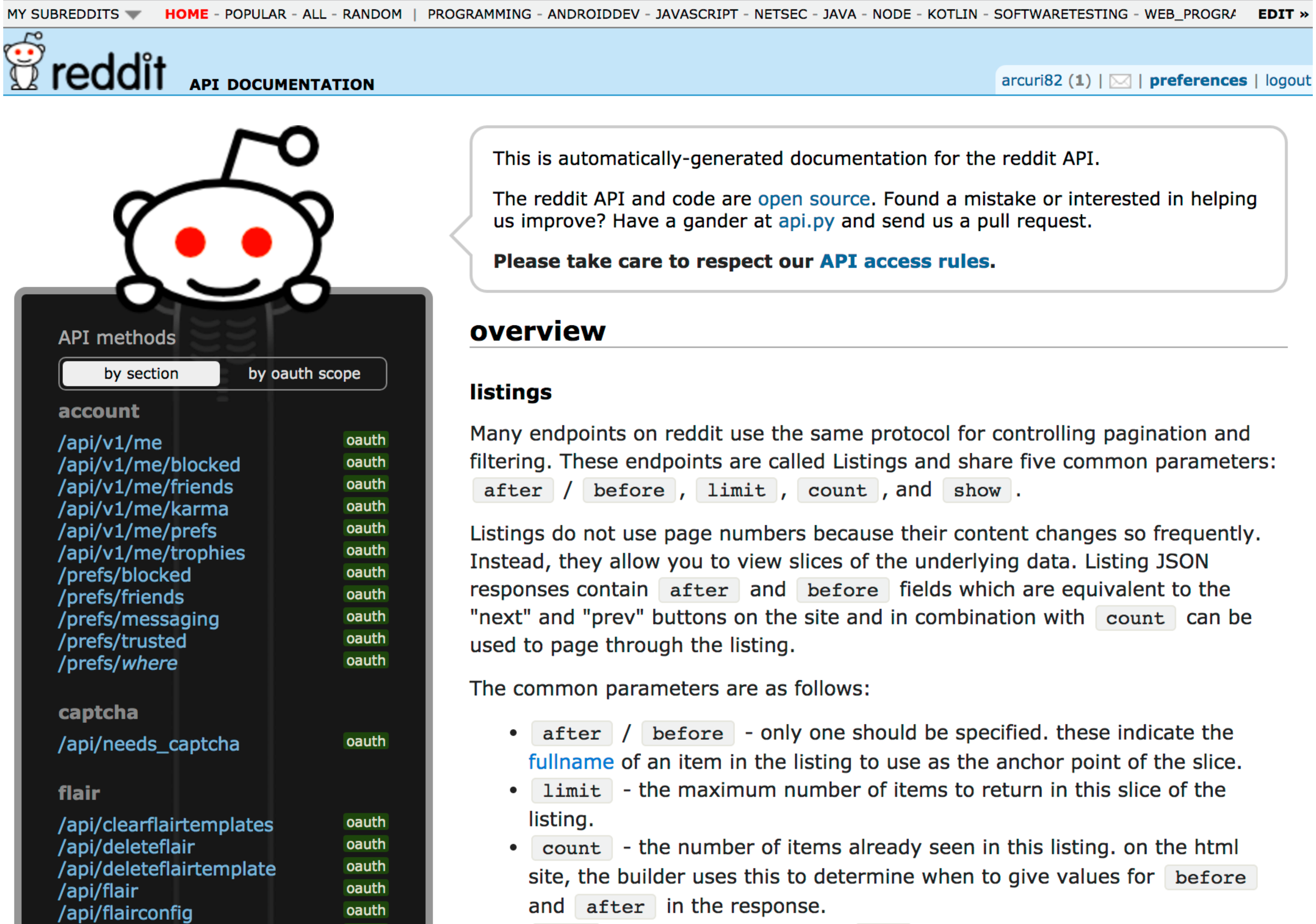
copy	`POST /files/fileId/copy`	Creates a copy of the specified file.



Getting started with the REST API

The foundation of all digital integrations with LinkedIn

The REST API is the heart of all programatic interactions with LinkedIn. All other methods of interacting, such as the JavaScript and Mobile SDKs, are simply wrappers around the REST API to provide an added level of convienence for developers. As a result, even if you are doing mobile or JavaScript development, it's still worth taking the time to familiarize yourself with how the REST API works and what it can do for you.


MY SUBREDDITS
HOME - POPULAR - ALL - RANDOM | PROGRAMMING - ANDROIDDEV - JAVASCRIPT - NETSEC - JAVA - NODE - KOTLIN - SOFTWARETESTING - WEB_PROGRA
EDIT »
reddit
API DOCUMENTATION
arcuri82 (1) | | preferences | logout

API methods

by section | by oauth scope

account

/api/v1/me	oauth
/api/v1/me/blocked	oauth
/api/v1/me/friends	oauth
/api/v1/me/karma	oauth
/api/v1/me/prefs	oauth
/api/v1/me/trophies	oauth
/prefs/blocked	oauth
/prefs/friends	oauth
/prefs/messaging	oauth
/prefs/trusted	oauth
/prefs/*where*	oauth

captcha

/api/needs_captcha	oauth

flair

/api/clearflairtemplates	oauth
/api/deleteflair	oauth
/api/deleteflairtemplate	oauth
/api/flair	oauth
/api/flairconfig	oauth

This is automatically-generated documentation for the reddit API.

The reddit API and code are open source. Found a mistake or interested in helping us improve? Have a gander at api.py and send us a pull request.

Please take care to respect our API access rules.

overview

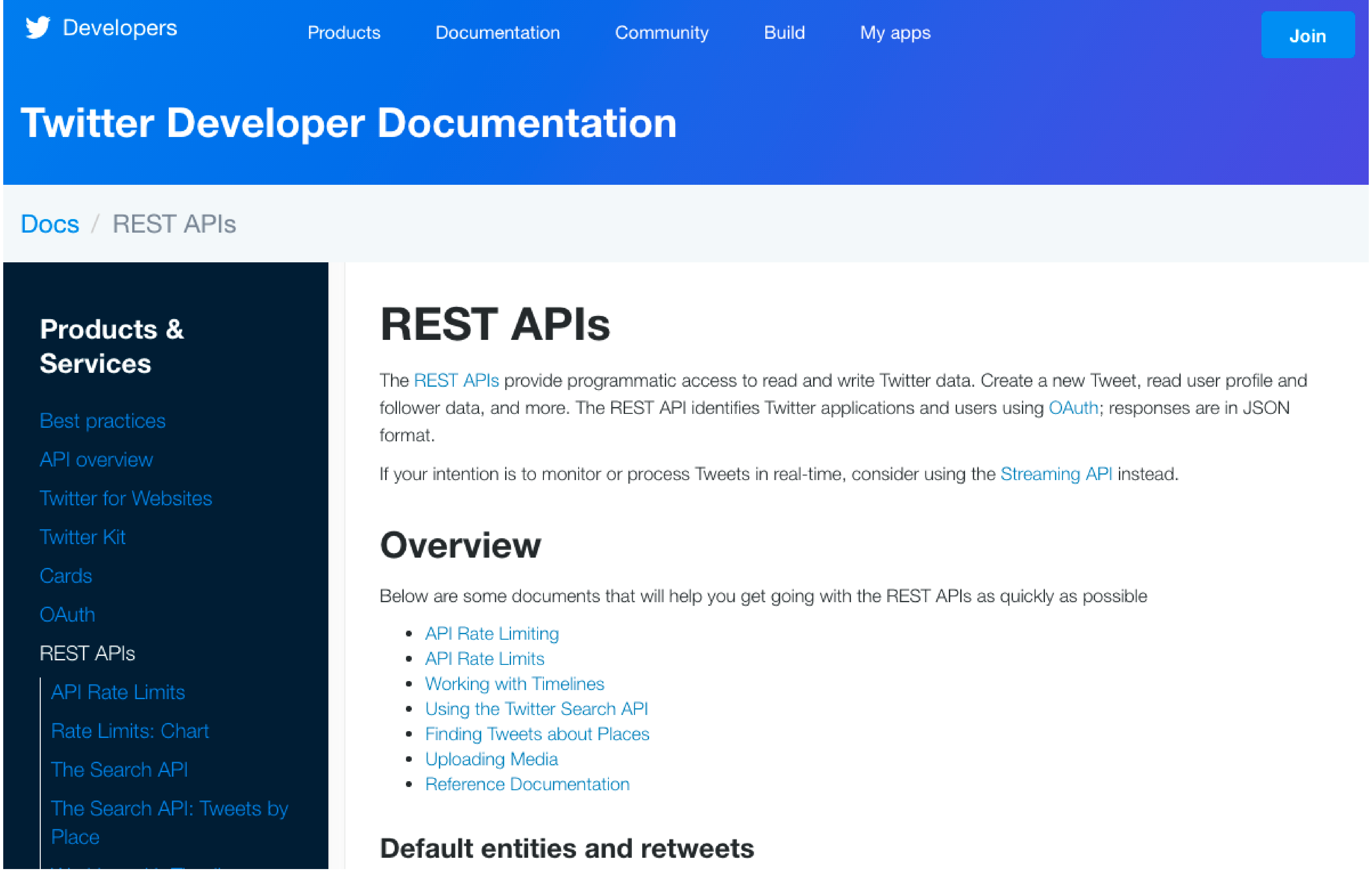
listings

Many endpoints on reddit use the same protocol for controlling pagination and filtering. These endpoints are called Listings and share five common parameters: `after` / `before`, `limit`, `count`, and `show`.

Listings do not use page numbers because their content changes so frequently. Instead, they allow you to view slices of the underlying data. Listing JSON responses contain `after` and `before` fields which are equivalent to the "next" and "prev" buttons on the site and in combination with `count` can be used to page through the listing.

The common parameters are as follows:

- `after` / `before` - only one should be specified. these indicate the fullname of an item in the listing to use as the anchor point of the slice.
- `limit` - the maximum number of items to return in this slice of the listing.
- `count` - the number of items already seen in this listing. on the html site, the builder uses this to determine when to give values for `before` and `after` in the response.



Twitter Developer Documentation

Docs / REST APIs

REST APIs

The REST APIs provide programmatic access to read and write Twitter data. Create a new Tweet, read user profile and follower data, and more. The REST API identifies Twitter applications and users using OAuth; responses are in JSON format.

If your intention is to monitor or process Tweets in real-time, consider using the Streaming API instead.

Overview

Below are some documents that will help you get going with the REST APIs as quickly as possible

- API Rate Limiting
- API Rate Limits
- Working with Timelines
- Using the Twitter Search API
- Finding Tweets about Places
- Uploading Media
- Reference Documentation

Default entities and retweets

RESTful APIs

RESTful APIs

- **Re**presentational **S**tate **T**ransfer (REST)
- Most common type of web services
- Access to set of resources using HTTP
- REST is *not a protocol*, but just architectural guidelines on how to define HTTP endpoints
    - Example: should not delete a resource when answering a GET, but no one will stop you from implementing an API that does that
- Introduced in a PhD thesis in 2000

REST Constraints

1. Uniform Interface
2. Stateless
3. Cacheable
4. Client-Server
5. Layered System
6. Code on demand (optional)

1: Uniform Interface

- Resource-based, identified by a URI
- The actual resource could be anything
  - e.g., rows in a SQL database, or image files on disk
- Client sees a *representation* of the resource, and the same resource can be given in different formats
  - eg, XML, JSON and TXT
- Hypermedia as the Engine of Application State (HATEOAS)
  - Resources connected by links… but hardly anyone uses it…

2: Stateless

- Resources could be stored in databases or files
- But the web service itself should be stateless
- This means that all info to process a request should come with the request itself
  - eg, as HTTP headers
- Consequence examples:
  - can restart process of web service at any time
  - horizontal scalability: can have 2 more instances of same service, does not matter which one is answering and in which order

3: Cacheable

- Cacheable: avoid making a request if previous retrieved data is still valid
- Very important for scalability
- Resources should define if they are cacheable or not, and how

4: Client–Server

- Clear cut between clients and servers
- Client only sees the URIs and the representation (eg JSON), but no internal details of server
    - eg does not even know if resource is stored in a database or on file
- Server does not know how data used on clients
- Consequence: clients and servers can be developed/updated independently, as long as URIs/representation are the same

5: Layered System

- For clients, should not matter if there is any intermediary on the way to the server
- Typical example: *reversed proxy*
  - eg, used for load balancing and access policy enforcement

6: Code on Demand (optional)

- Servers can temporarily extend or customize the functionality of a client by transferring executable code
  - eg, transfer JavaScript code
- Among the constraints that define REST, this is optional

The Term “REST”

- Most APIs out there are called REST by their developers…
- … but “technically”, they aren’t
- For example, nearly *no one* uses HATEOAS
- So, nowadays, REST loosely means: “*A web API where resources are hierarchically structured with URIs, and operations follow the semantics of the HTTP verbs/methods*”

Example for a Product Catalog

- Full URLs, eg **www.foo.com/products**
- GET **/products**
  - (return all available products)
- GET **/products?k=v**
  - (return all available products filtered by some custom parameters)
- POST **/products**
  - (create a new product)
- GET **/products/{id}**
  - (return the product with the given id)
- GET **/products/{id}/price**
  - (return the price of a specific product with a given id)
- DELETE **/products/{id}**
  - (delete the product with the given id)

Resource Hierarchy

- Consider the resource: **/users/3457/items/42/description**
- **/users**: resource representing a set of users
- **/3457**: a specific user with that given id among the set of users */users*
- **/items**: a set of items belonging to the user 3457
- **/42**_:_ a specific item with id 42 that the user 3457 owns
- **/description**: among the different properties/fields of item 42, just consider its *description*

Cont.

- *GET **/users/3457/items/42/description***
- It means: retrieve the description of item with id 42, which belongs to the user with id 3456
- But what about *GET **/items/42/description*** ???
- “Technically”, they would be 2 *different* resources, because there are two different URIs
- But in practice, they are the same

Backend Representation

- ***/users/3457/items/42/description***
- Could be two different tables in a SQL database, eg *Users* and *Items*
- Or could be a single JSON file on disk…
- or the REST API just collects such data from two other different web services…
- or whatever you fancy…
- Point is, for the client this does not matter at all!

Available URIs

- 1st) *GET **/users/3457/itemIds***
- 2nd) *GET **/items/42/description***
- It means: first retrieve the ids of all items belonging to user 3457. Then, to get description for a specific one of them with id 42, make a second GET
- But in the 2nd GET, what if we rather used ***/users/3457/items/42/description*** ???

Cont.

- 1st) *GET* ***/users/3457/itemIds***
- 2nd) *GET* ***/items/42/description***
- 3rd) *GET* ***/users/3457/items/42/description***
- Whether the 2nd or the 3rd (or both) endpoint is needed depends on how clients will typically interact with the API
    - do they need to access to items regardless of their user owners?
- Point is: you need to *implement* a handler for each endpoint

Path Elements

- ***/users/3457/items/42/description***
- How does a client know that ***/users*** and ***/items*** are collections/sets but not ***/description*** ?
- “Technically”, each of those tokens are path elements, with no specific semantics
- Client has to read the documentation of the API
- However, to make things simpler, it is a convention to use *plural* names for set resources

Resource Filtering

- Assume you want to retrieve all users that are in Norway
- 1st) *GET* ***/users/inNorway***
- Problem is, what if you still want to retrieve single users by id?
- 2nd) *GET* ***/users/{id}***
  - Where {} just represents a variable matching any single path element input
- Ambiguity: here ***/users/inNorway*** would be matched by both endpoints
  - ie, *inNorway* could be treated as a user id

Cont.

- 1st) *GET* ***/users/inNorway***
- 2nd) *GET* ***/users/byId/{id}***
- Here there would be no ambiguity, but…
- … what would be the semantics of the intermediate resource ***/users/byId*** ???
- Paths in the URIs should represent resources, and not actions on them

Cont.

- 1st) *GET* ***/users?country=norway***
- 2nd) *GET* ***/users/{id}***
- When we want to apply a filter to get a subset of a collection, then we use *query parameters*
  - recall URIs: start with "?", followed by pairs <key>=<value>
- Extra benefit: we can later add extra filter options (e.g., *ageMin=18*), without altering the routing of requests to the endpoint ***/users***

Resource Creation

- *POST* ***/users***
  - POST operation on a collection
  - Payload used to create new element added to the collection
  - Response will have *Location* HTTP header telling where to find the newly created resource, eg *Location:/users/42*
- *PUT* ***/users/42***
  - PUT operation directly on the URI of the new resource
  - Need to specify id

Cont.

- Which one to use? POST or PUT?
- When id is chosen by server (eg linked to an id from SQL database), you need POST
- If you use PUT, client must choose the id, and it must be *unique*
  - otherwise, you would just overwrite an existing resource

PUT vs POST

- *1st) GET /users/42* => Response 404
- *2nd) PUT /users/42*
- This would make no sense, because:
    1. Not going to do hundreds of GETs until find one with 404 Not Found
    2. Two HTTP requests in sequence are not necessarily atomic, eg, before PUT is executed, someone else could have create the resource, and you would just then overwrite it

Cont.

- 1st) *POST* ***/users*** *=>* Location: */users/42*
- 2nd) *PUT* ***/users/42/address***
- Assume you create a new user with a POST operation, but without an *address*
- You could then want to create the *address* resource directly by using a PUT
    - point is that the resource does not have an id in itself, but rather the id is in a path element ancestor
- However, most of the time you would not expose each single field of an object as its own URI endpoint, but rather do a PATCH
    - eg, *PATCH /users/42*

Resource Representation

- 1st) *GET /users/42*
- 2nd) *GET /users/42**.json***
- 3rd) *GET /users/42**.xml***
- For what you know, the REST service could store users in a SQL database or a CSV file
- What you get is a *representation* of a resource, which can be in different formats, based on client's needs
- But what's the problem here?

Cont.

- 1st) *GET /users/42*
- 2nd) *GET /users/42**.json***
- 3rd) *GET /users/42**.xml***
- Because the URIs are different, they are technically 3 *different* resources
  - whether they map to the same entity on the backend is another story…
- A URI has no concept of type: adding a “*.json*” extension does NOT change the semantics

Cont.

- *GET* ***/users/42***
- You should avoid type extensions on your resources
  - although you might see many APIs doing it…
- Choosing among different types should be based on HTTP headers like *Accept*
  - eg, "*Accept: application/json*"
- If a client asks for a specific representation (eg XML), that does not mean that the server would support it
- If Accept missing, or generic **/**, server would just use the default representation (e.g., JSON)

Static and Dynamic Resources

Static Resources

- HTML files
  - usually just a single *index.html* is SPAs
- CSS
- JavaScript source files
  - eg, *bundle.js*
- Images, documents, or any type of files to download
  - eg, PDFs
- etc.

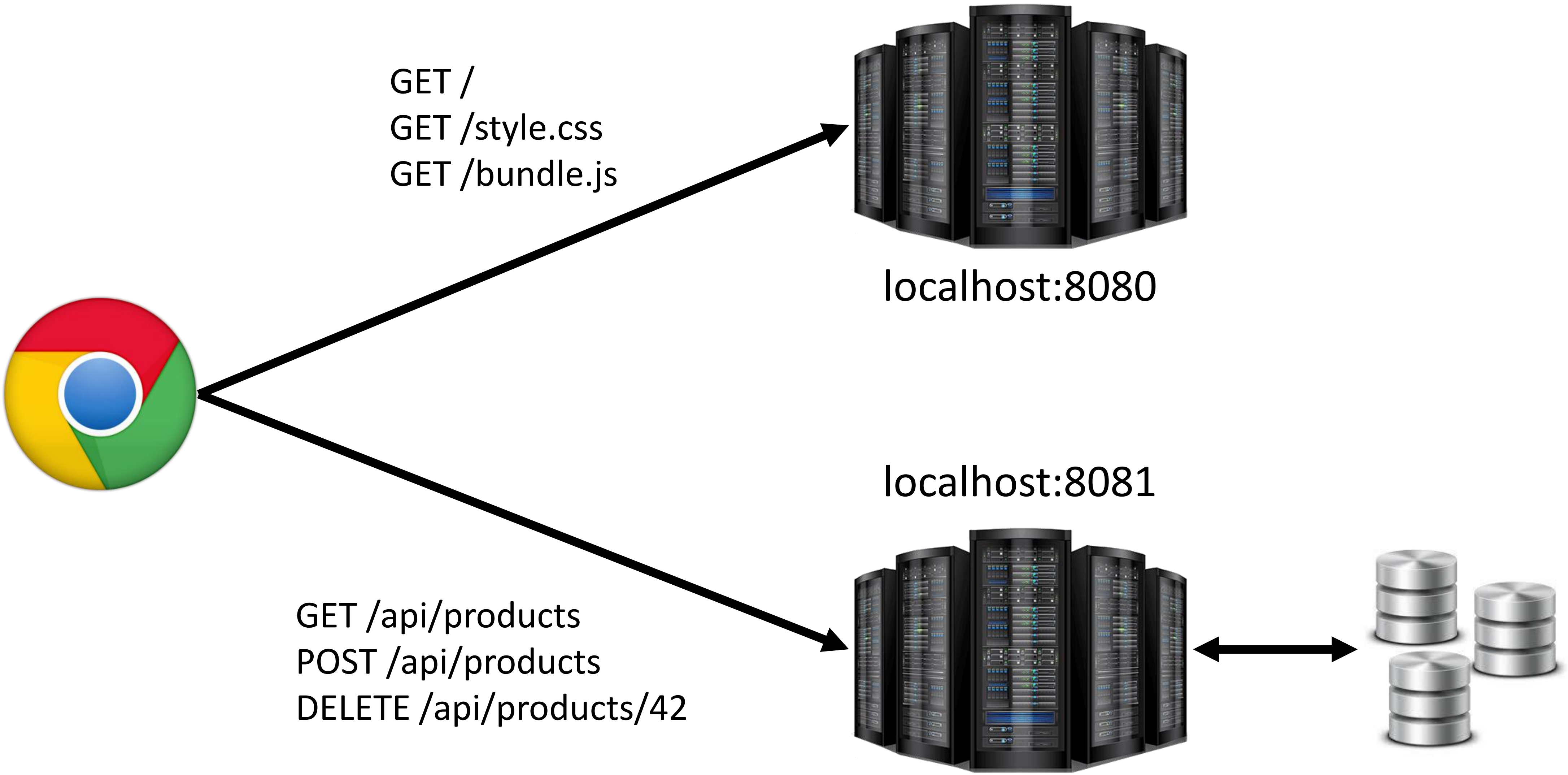
Dynamic Resources

- Data that usually change through time
- Can depend on user interactions
  - create an account, add items to a shopping cart, etc.
- Could have long term storage in SQL and NoSQL databases
- In REST, usually we will get a *representation* of those resource in JSON format
- Handling/generation of dynamic resources usually depend on *business logic* in the so called *backend*

Static vs Dynamic

- “Usually”, the static resources will define the *frontend* of a web application
    - HTML/CSS/JS/images/etc.
- The *backend* will be a server providing data via JSON, and long term storage with databases
- *Backend* will be a process with business logic written in some programming language (JS, or Java and C#)
- Still, for both static and dynamic resources going to use HTTP

- *Frontend* (e.g., React app) static assets still need to be provided by a HTTP server
- *Backend* (e.g., REST API) with business logic and access to database is still a HTTP server



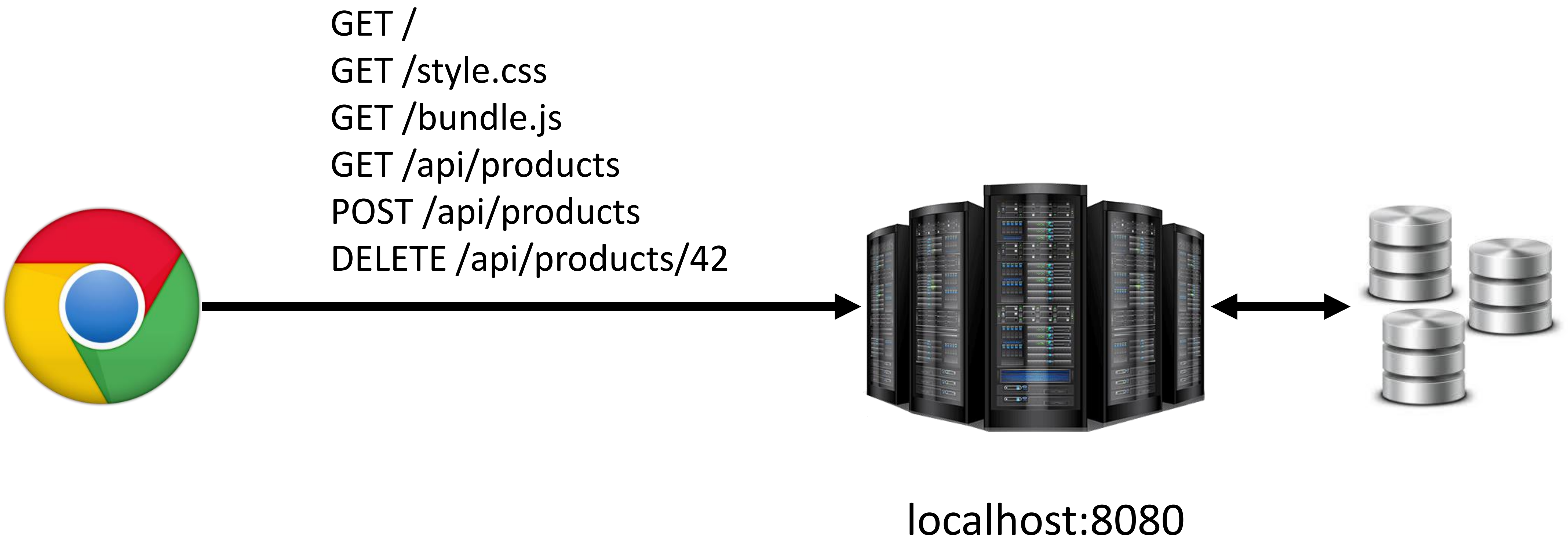
GET /
GET /style.css
GET /bundle.js

localhost:8080

localhost:8081

GET /api/products
POST /api/products
DELETE /api/products/42

- From point of view of the browser, no difference between static assets and JSON responses from a REST API
    - still going to use HTTP
- *Frontend* and *backend* can be handled by the same HTTP server
    - this also avoid issues with CORS (discussed in more details in next class)



GET /
GET /style.css
GET /bundle.js
GET /api/products
POST /api/products
DELETE /api/products/42

localhost:8080