REST APIs

Slides by Andrew DeOrio

Agenda

- HTTP review
- REST API intro
- JSON
- Tools
- Collections and pagination
- Verbs and status codes
- Design principles

Review: HTTP request methods

- Request method indicates server action
- GET: request a resource
  - Example: load a page
- HEAD: identical to GET, but without response body
  - Example: see if page has changed
- POST: send data to server
  - Example: web form

- Others we will cover later in the REST API lecture
  - TODAY

Review: HTTP request headers

- Headers accompany request
- Most are optional

```
$ curl --verbose http://cse.eecs.umich.edu/ >
index.html
* Connected to cse.eecs.umich.edu
(141.212.113.143) port 80 (#0)
> GET / HTTP/1.0
> Host: cse.eecs.umich.edu
> User-Agent: curl/7.54.0
> Accept: */*
```

- `Host` distinguishes between DNS names sharing a single IP address
  - Required as of HTTP/1.1
- `User-Agent:` which browser is making the request
- `Accept:` which content ("file") types the client will accept

Review: HTTP status code

- Response starts with a status code
  - 1XX: Informational
  - 2XX: Successful
  - 3XX: Client Error
  - 4XX: Server Error

-
  ```
  $ curl --verbose http://cse.eecs.umich.edu/
  > GET / HTTP/1.0
  < HTTP/1.0 200 OK
  ```

-
  ```
  $ curl --verbose http://cse.eecs.umich.edu/asdf
  > GET /asdf HTTP/1.0
  < HTTP/1.0 404 Not Found
  ```

Review: HTTP response headers

- Headers accompany a response
- Most are optional

```
$ curl --verbose http://cse.eecs.umich.edu/
* Connected to cse.eecs.umich.edu
> GET / HTTP/1.0
...
< HTTP/1.0 200 OK
< Date: Tue, 12 Sep 2017 20:04:20 GMT
< Server: Apache/2.2.15 (Red Hat)
< Accept-Ranges: bytes
< Connection: close
< Transfer-Encoding: chunked
< Content-Type: text/html; charset=UTF-8
```

Review: HTTP content type

- Content type describes the "file" type and encoding

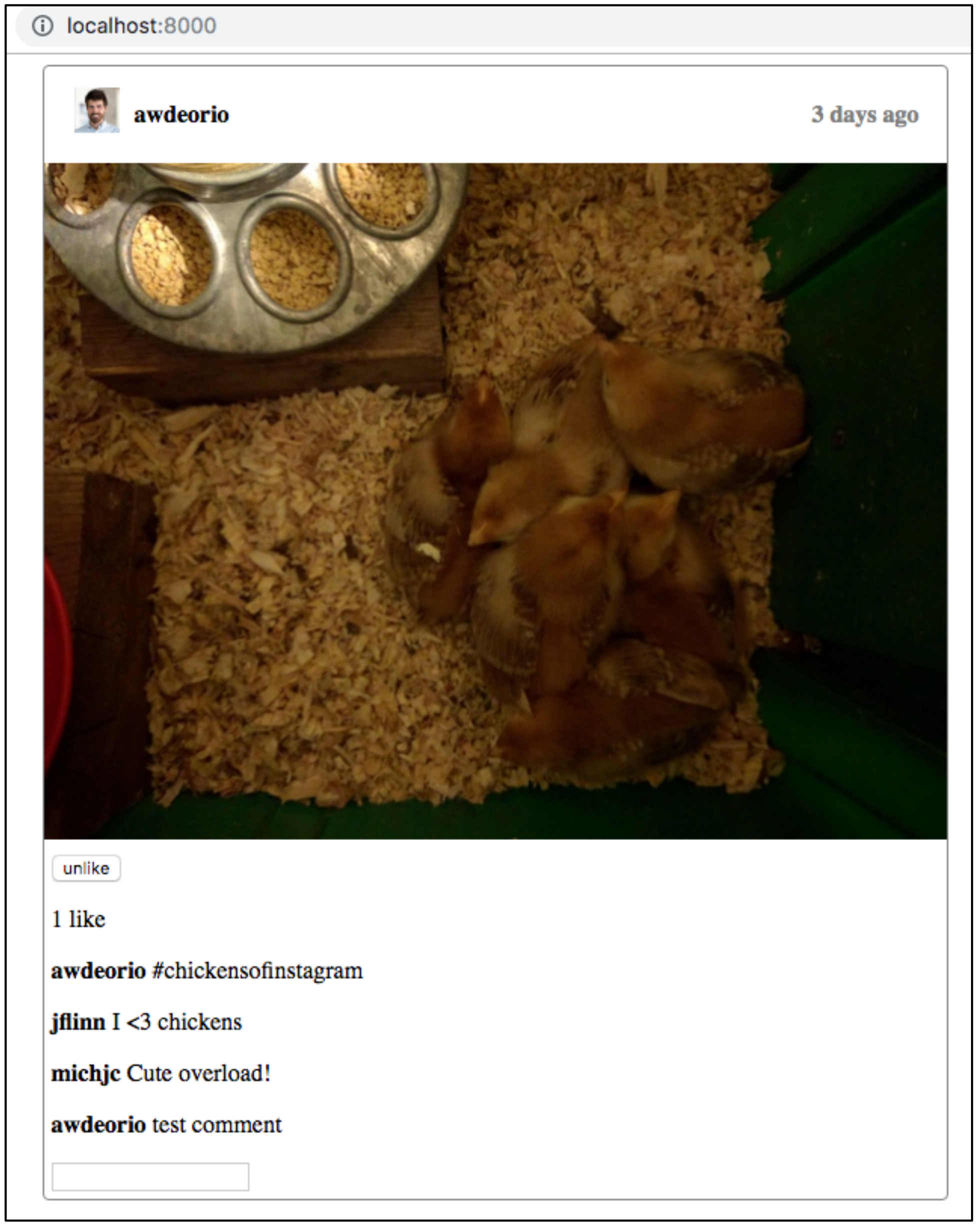
```
$ curl --verbose http://cse.eecs.umich.edu/
* Connected to cse.eecs.umich.edu
> GET / HTTP/1.0
...
< HTTP/1.0 200 OK
...
< Content-Type: text/html; charset=UTF-8

<!doctype html><html lang="en">
...
</html>
```

- In past lectures, GET requests have returned HTML content
- In today's lecture, we'll return JSON data

Agenda

- HTTP review
- **REST API intro**
- JSON
- Tools
- Collections and pagination
- Verbs and status codes
- Design principles

Not a REST API: human-readable



localhost:8000

awdeorio 3 days ago

unlike

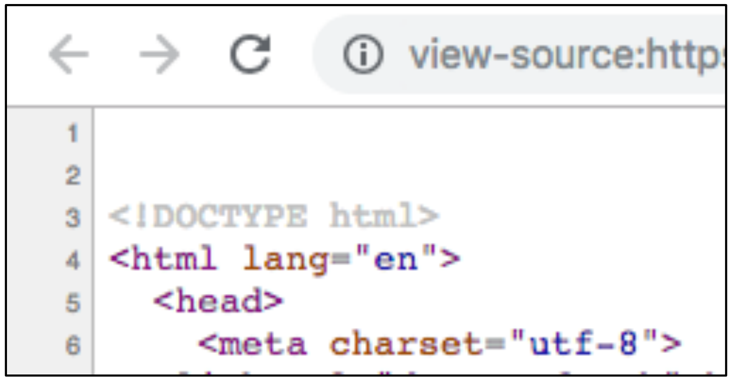
1 like

awdeorio #chickensofinstagram

jflinn I <3 chickens

michjc Cute overload!

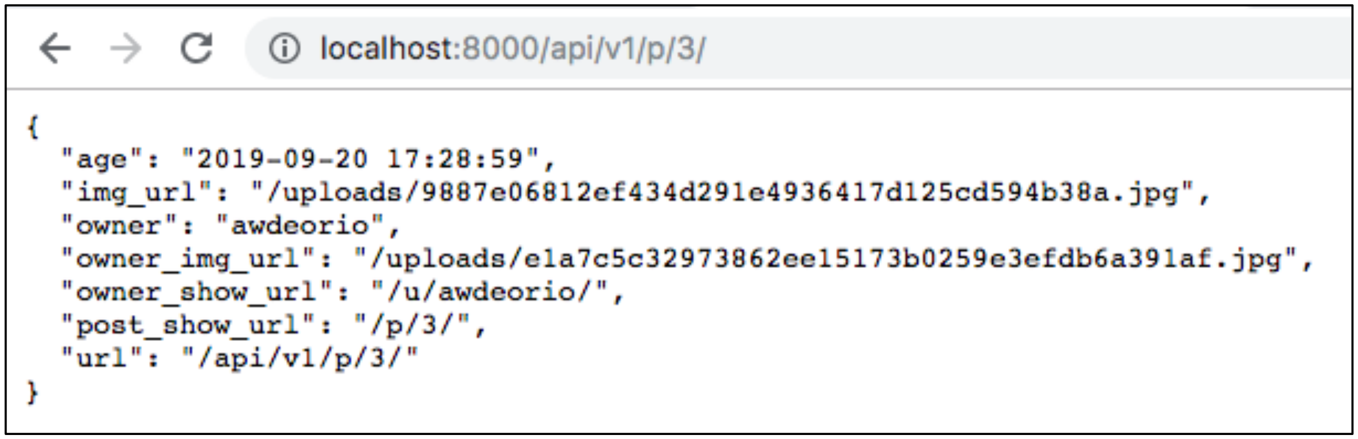
awdeorio test comment



view-source:http

```


<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
```

REST API: machine-readable

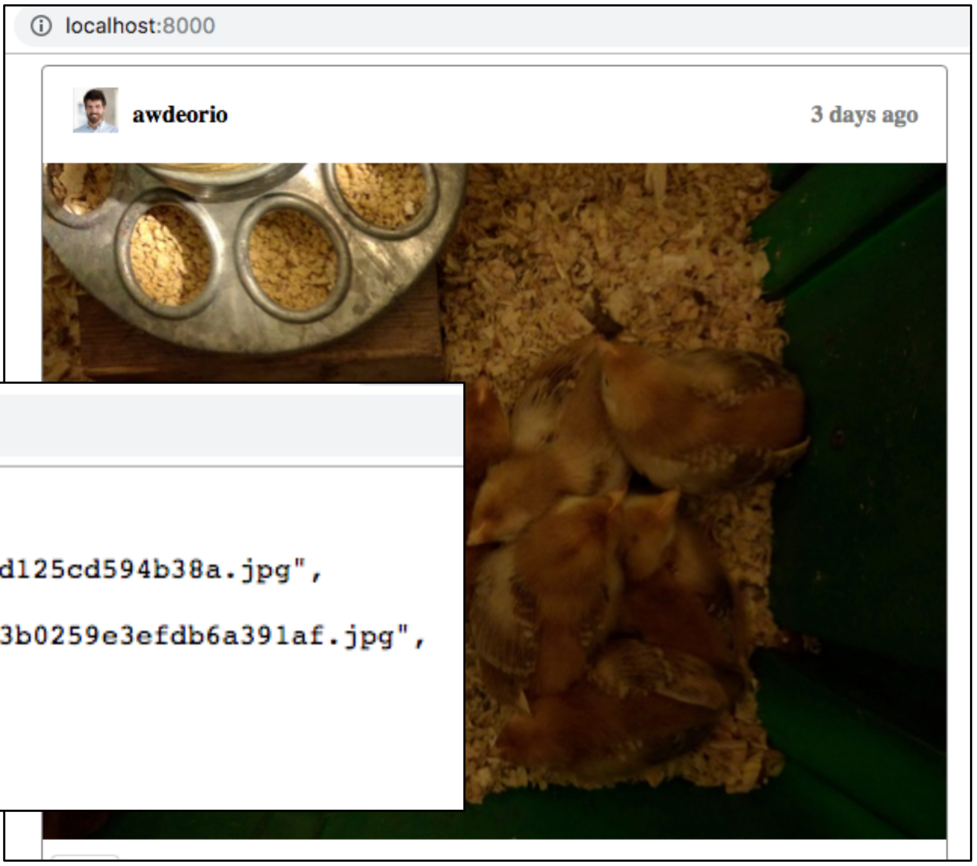


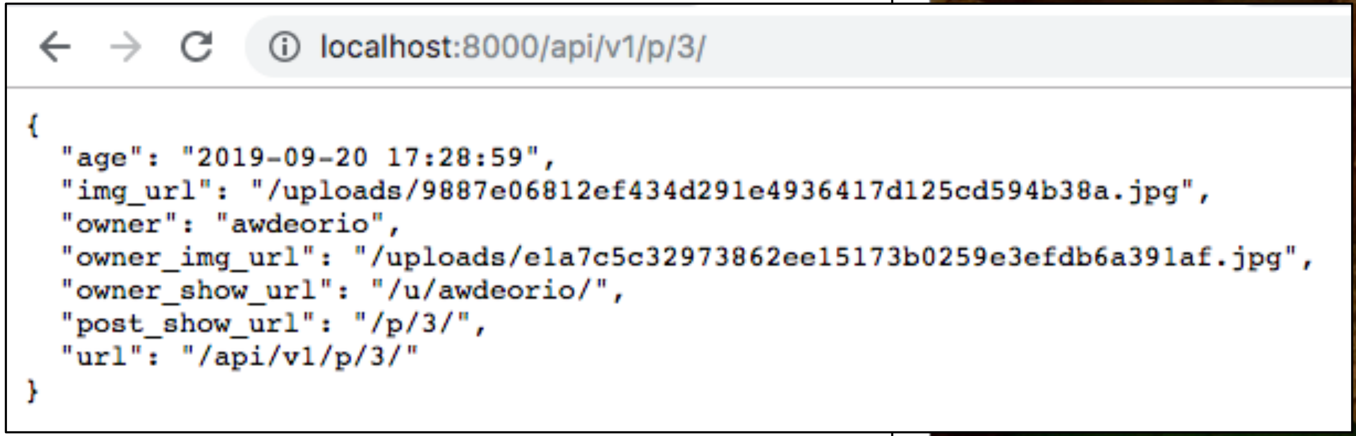
```
{
  "age": "2019-09-20 17:28:59",
  "img_url": "/uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg",
  "owner": "awdeorio",
  "owner_img_url": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  "owner_show_url": "/u/awdeorio/",
  "post_show_url": "/p/3/",
  "url": "/api/v1/p/3/"
}
```

REST API: machine-readable

- JavaScript runs in browser
- JavaScript receives JSON data from server
- JavaScript renders data on page

- Also: Two servers can communicate via JSON.




localhost:8000/api/v1/p/3/

```
{
  "age": "2019-09-20 17:28:59",
  "img_url": "/uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg",
  "owner": "awdeorio",
  "owner_img_url": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  "owner_show_url": "/u/awdeorio/",
  "post_show_url": "/p/3/",
  "url": "/api/v1/p/3/"
}
```

REST APIs use HTTP

- HTTP request includes a method
- HTTP response includes a status code and JSON data

```
$ curl --verbose localhost:8000/api/v1/p/1/
> GET /api/v1/p/1/ HTTP/1.0
< HTTP/1.0 200 OK
< Content-Type: application/json
{
  "age": "2019-09-20 17:28:59",
  "img_url": "/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  "owner_img_url": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  ...
}
```

REST APIs

- REST: Representational State Transfer
- Interoperability between different web systems

- REST is not ...
  - A standard
  - A language

- REST is ...
  - A collection of principles
  - Some best practices
  - Usually uses HTTP and JSON

- Originally defined by Roy Fielding in his doctoral dissertation

Agenda

- HTTP review
- REST API intro
- **JSON**
- Tools
- Collections and pagination
- Verbs and status codes
- Design principles

JSON

- JSON: JavaScript Object Notation
- Lightweight data-interchange format
- Based on JavaScript syntax
    - Uses conventions familiar to programmers in many languages
- Commonly used to send data from a server to a web client
    - Client parses JSON using JavaScript and displays content
- Ubiquitous with REST APIs

JSON structures

- Object: a collection of name/value pairs
    - In other languages: object, record, struct, dictionary, hash table, keyed list, or associative array
      `{ "name": "DeOrio", "num_chicken": 4 }`
- Array: an ordered list of values
    - In other languages: array, vector, list, or sequence
      `[ "Marilyn", "Maude", "Myrtle II", "Mabel"]`
- A value is:
    - string
    - number
    - `true`
    - `false`
    - `null`
    - Object
    - Array

Valid JSON

- Validate JSON

```
$ curl -s https://api.github.com/users/awdeorio | jsonlint
```

- Pitfall: **no trailing commas allowed!**

```
{
  "login":"awdeorio",
  "id":7503005,
  ...


   "updated_at": "2017-12-12T19:11:17Z” /  ,

}
```

- More details: http://www.json.org/

Agenda

- HTTP review
- REST API intro
- JSON
- **Tools**
- Collections and pagination
- Verbs and status codes
- Design principles

curl

- REST API at the command line
- HTTP `GET` request returns a JSON-formatted string

```
$ curl https://api.github.com/users/awdeorio
{
  "login": "awdeorio",
  "id": 7503005,
  ...
```

jq and python

- Pretty-print JSON using `jq`

```
$ curl -s https://api.github.com/users/awdeorio | jq
{
  "login":"awdeorio",
  "id":7503005,
  ...
```

- Pretty-print JSON using Python

```
$ curl -s https://api.github.com/users/awdeorio | python -m json.tool
{
  "login":"awdeorio",
  "id":7503005,
  ...
```

Httpie

- Improved CLI and color coding with httpie

```
$ http https://api.github.com/users/awdeorio
HTTP/1.0 200 OK
{
  "login":"awdeorio",
  "id":7503005,
  ...
```

httpbin.org

- https://httpbin.org is an echo server
    - Responds with whatever you sent to it

```
$ http POST httpbin.org/anything hello=world
...
{
    ...
    "json": {
        "hello": "world"
    },
    "method": "POST",
    "url": "http://httpbin.org/anything"
}
```

Agenda

- HTTP review
- REST API intro
- JSON
- Tools
- **Collections and pagination**
- Verbs and status codes
- Design principles

Detail view AKA item view

- Our previous example showed a REST API *detail view* or *item view*
- It returns one object from the database
- Notice the the *id* part of the URL
  - Also called a "slug"

```
$ curl localhost:8000/api/v1/p/1/
{
  "age": "2019-09-20 17:28:59",
  "img_url": "/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  "owner_img_url": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  ...
  "url": "/api/v1/p/1/"
}
```

List view AKA collection view

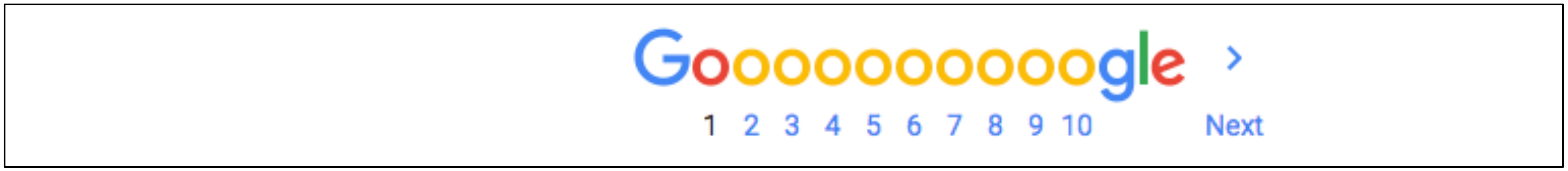
- REST APIs often expose collections of items

```
$ curl localhost:8000/api/v1/p/
{
  "results": [
    {
      "postid": 3,
      "url": "/api/v1/p/3/"
    },
    {
      "postid": 2,
      "url": "/api/v1/p/2/"
    },
    ...
  ]
}
```

Pagination

- Pagination from the UI perspective
- REST API enables this
- Instagram et. al use REST API pagination for infinite scroll

...



Pagination

- List views should return a limited number of items
  - What if there were 10 million posts?
- Sensible default, e.g., 10 posts
  - `$ curl localhost:8000/api/v1/p/`
- Get the next 10 results
  - `$ curl localhost:8000/api/v1/p/?page=1`
- Customizable size
  - `$ curl localhost:8000/api/v1/p/?size=20`

Agenda

- HTTP review
- REST API intro
- JSON
- Tools
- Collections and pagination
- **Verbs and status codes**
- Design principles

REST API verbs and status codes

- All our examples so far have been GET requests
  - GET request
  - 200 OK response

- This takes care of reading data. What about create, modify and delete?

REST API verbs

- GET: return datum
  - Example: return a post
- POST: create new datum
  - Example: create a new post
- PATCH: update part of a datum
  - Example: modify part of an existing post
- PUT: replace the entire datum
  - Example: replace an existing post
- DELETE: delete datum
  - Example: remove a post

POST request

- `POST` creates an object
- Request includes JSON body

```
POST localhost:8000/api/v1/p/ HTTP/1.0
{
  "img_url": "122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  ...
}
```

POST response

- `POST` returns `201 CREATED` on success
- Response includes a copy of the created object
  - Object usually includes a link to itself

```
POST localhost:8000/api/v1/p/ HTTP/1.0
...
HTTP/1.0 201 CREATED
{
  "img_url": "122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  ...
  "url": "/api/v1/p/1/"
}
```

PATCH request

- `PATCH` modifies part of an existing object
- Request URL includes an ID
- Request includes JSON body

- Example: change the picture in a post
- Notice that the JSON body is short, and only contains the field that should be modified

```
PATCH localhost:8000/api/v1/p/1/ HTTP/1.0
{
  "img_url": "ad7790405c539894d25ab8dcf0b79eed3341e109.jpg",
}
```

PATCH response

- `PATCH` returns `200 OK` on success
- Response includes a copy of the **entire** modified object

```
PATCH localhost:8000/api/v1/p/1/ HTTP/1.0
...
HTTP/1.0 200 OK
{
  "img_url": "ad7790405c539894d25ab8dcf0b79eed3341e109.jpg",
  "owner": "awdeorio",
  ...
  "url": "/api/v1/p/1/"
}
```

PUT request

- `PUT` replaces an entire existing object
- Request URL includes an ID
- Request includes JSON body
- Example: replace an entire post
- The JSON body is long, and contains a replacement value for every field

```
PUT localhost:8000/api/v1/p/1/ HTTP/1.0
{
  "img_url":  ...,
  "owner": "jflinn",
  "owner_img_url": ...,
...
}
```

PUT response

- `PUT` returns `200 OK` on success
- Response includes a copy of the **entire** modified object

```
PUT localhost:8000/api/v1/p/1/ HTTP/1.0
...
HTTP/1.0 200 OK
{
  "img_url":  ...,
  "owner": "jflinn",
  "owner_img_url": ...,
...
}
```

DELETE request

- `DELETE` removes an object
- Request URL includes an ID
- No body in request

```
DELETE localhost:8000/api/v1/p/1/ HTTP/1.0
```

DELETE response

- `DELETE` returns `204 NO CONTENT` on success
- No body in response

```
DELETE localhost:8000/api/v1/p/1/ HTTP/1.0
...
HTTP/1.0 204 NO CONTENT
```

Not found response

- `GET` a deleted item, receive a 404 response

```
DELETE localhost:8000/api/v1/p/1/ HTTP/1.0
HTTP/1.0 204 NO CONTENT

GET localhost:8000/api/v1/p/1/ HTTP/1.0
HTTP/1.0 404 NOT FOUND
```

REST API status codes

- 200 OK
- 201 Created
  - Successful creation after POST
- 204 No Content
  - Successful DELETE
- 304 Not Modified
  - Used for conditional GET calls to reduce band-width usage
- 400 Bad Request
  - General error
- 401 Unauthorized
  - Missing or invalid authentication
- 403 Forbidden
  - User is not authorized
- 404 Not Found
  - Resource could not be found
- 409 Conflict
  - E.g., duplicate entries and deleting root objects when cascade-delete is not supported
- 500 Internal Server Error
  - General catch-all for server-side exceptions

http://www.restapitutorial.com/httpstatuscodes.html

Agenda

- HTTP review
- REST API intro
- JSON
- Tools
- Collections and pagination
- Verbs and status codes
- **Design principles**

REST design principles

- Uniform interface
  - Resource-based
  - Manipulation of resources through representations
  - Self-descriptive messages
  - Hypermedia as the engine of application state (HATEOAS)
- Client-server architecture
- Stateless
- Cacheable
- Layered
- Code on demand (optional)

Uniform interface: resource-based

- Individual resources are identified in requests using URIs as resource identifiers.
- Think of a URI like a pointer
- ID in the URL

```
GET /api/v1/p/1/ HTTP/1.0
```

Uniform interface: manipulation of resources through representations

- When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the server.
- Object usually contains a link to itself

```
GET /api/v1/p/1/ HTTP/1.0


HTTP/1.0 200 OK
{
  ...,
  "url": "/api/v1/p/1/"
}
```

Uniform interface: self-descriptive messages

- Each message includes enough information to describe how to process the message
- Content-Type and charset

```
GET /api/v1/p/1/ HTTP/1.0


HTTP/1.0 200 OK
> Content-Type:application/json; charset=utf-8
...
{
   ...
}
```

Uniform interface: HATEOAS

- HATEOAS: Hypermedia as the Engine of Application State
- Clients deliver state via body contents, query-string parameters, request headers and the requested URI (the resource name).
- Services deliver state to clients via body content, response codes, and response headers. This is technically referred-to as hypermedia (or hyperlinks within hypertext).
- Everything you need is in the request

```
POST /api/v1/p/ HTTP/1.0
{
  "img_url": "122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  ...
}
```

Uniform interface: HATEOAS

- Links are contained in the returned body (or headers) to supply the URI for retrieval of the object itself or related objects.

```
GET /api/v1/p/ HTTP/1.0
HTTP/1.0 200 OK
{
  "results": [
    {
      "postid": 3,
      "url": "/api/v1/p/3/"
    },
    {
      "postid": 2,
      "url": "/api/v1/p/2/"
    },
     ...
```

Client-server architecture

- The uniform interface separates clients from servers
- *Abstraction* between client and server

- Can change the server without modifying the client
- Can change the client without modifying the server

- Example: database is too slow, replace portions of it with key-value store like Redis

Stateless

- Everything needed to handle the request is in the request itself
  - URI, query-string parameters, body, or headers
- After the server does it's processing, the appropriate state, or the piece(s) of state that matter, are communicated back to the client via headers, status and response body

```
POST /api/v1/p/ HTTP/1.0


HTTP/1.0 201 Created
{
  // contents of created object
}
```

Cacheable

- Clients cache some kinds of responses to eliminate requests
    - Example: cache an image so you don't load it every time
- Responses must implicitly or explicitly define themselves as cacheable
- Example: `Last-Modified` header

Layered

- A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way
- No need to connect to a specific machine
- Just need the data from this URI

Code on demand (optional)

- Servers are able to temporarily extend or customize the functionality of a client by transferring logic to it that it can execute
- Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript
- Not common

Summary

- A client and a server can communicate via a REST API
- Two servers can communicate via a REST API
- REST APIs use HTTP
- REST APIs are machine-readable
- REST APIs usually return JSON data

Public APIs

- GitHub
  https://developer.github.com/v3/
- LinkedIn
  https://developer.linkedin.com/
- Facebook
  https://developers.facebook.com/docs/graph-api
- Twitter
  https://dev.twitter.com/rest/public