

Working With Web APIs

Computer Science and Engineering ■ College of Engineering ■ The Ohio State University

Lecture 13

Passing arguments in HTTP: GET

- Arguments are key-value pairs

Mascot: Brutus Buckeye

Dept: CS&E

- Encoded in the *query string* of a URL

scheme://FQDN:port/path?**query**#fragment

- Encoding: **application/x-www-form-urlencoded**

- Key-value pairs are separated by **&** (or **;**)
- Each key is separated from value by **=**
- Replace spaces with **+** (arcane!)
- Then normal URL encoding (using **%hh**)

Mascot**=**Brutus**+**Buckeye**&**Dept**=**CS**%26**E

Examples

□ Wikipedia search

`http://en.wikipedia.org/w/index.php?
search=ada+lovelace`

□ OSU news articles

`https://news.osu.edu/?
q=Goldwater&search.x=1&search.y=0`

□ Random passwords from random.org

`https://random.org/passwords/?
num=5&len=8&format=plain`

- Demo: use Chrome dev tools to "Copy as cURL"
- See guidelines and [API](#) for http clients

Passing Arguments in HTTP: POST

- ❑ Encoded as part of the request *body*
- ❑ Advantages over query string:
 - Arbitrary length (URL length is limited)
 - Arguments are not saved in browser history
 - Result is not cached by browser
 - Arguments are less likely to be accidentally shared (eg not part of URL, won't be bookmarked), so slightly more secure
- ❑ Content-Type indicates encoding of body
 - **application/x-www-form-urlencoded**
 - ❑ Same encoding as used with GET and query string
 - **multipart/form-data**
 - ❑ Better for binary data than urlencoding (where 1 byte → 3 bytes)
 - More options too:
 - ❑ **application/xml**, **application/json**, ...

Passing Arguments: GET vs POST

□ GET

GET /passwords/?num=5&len=8&format=plain HTTP/1.1

Host: www.random.org

□ POST

POST /passwords/ HTTP/1.1

Host: www.random.org

Content-Type: application/x-www-form-urlencoded

Content-Length: 24

num=5&len=8&format=plain

Passing Arguments: Summary

- Arguments in GET requests
 - Uses request query string
 - Limited length, highly visible (eg in location bar)
 - Encoded with `application/x-www-form-urlencoded`
- Arguments in POST requests
 - Uses request body
 - No size limit, not bookmarked
 - Choice for content type (ie encoding), most commonly:
 - `application/x-www-form-urlencoded`
 - `multipart/form-data`
 - `application/json`

JSON

- ❑ JavaScript Object Notation
- ❑ String-based representation of a value
 - Serialization: converting value -> string
 - Deserialization: converting string -> value
- ❑ Easy enough for people to read
- ❑ Really designed for computers to parse
 - The *lingua franca* for transfer of (object) values via HTTP
 - Used both ways: request arguments and responses
- ❑ MIME type: application/json

JSON Has Six Data Types

- ❑ Text: a string, "..."
`"hello", "I said \"hi\"", "3.4", ""`
- ❑ Number: integer or floating point
`6, -3.14, 6.022e23`
- ❑ Boolean
`true, false`
- ❑ Null
`null`
- ❑ Array: ordered list of values, [...]
`[3, 2, 1, "go"], [[1, 3], [7, -2]]`
- ❑ Object: set of name/value pairs, {...}
 - Names are text
 - Values are any JSON type (text, number, array, object...)
`{"mascot": "Brutus", "age": 19, "nut": true}`

Example

```
{"current_page":1,"limit":20,"next_page":1,"previous_page":1,"results":[{"id":"G1GBIY0wAAAd","joke":"How much does a hipster weigh? An instagram."},{ "id":"xc21Lmbxcib","joke":"How did the hipster burn the roof of his mouth? He ate the pizza before it was cool."}], "search_term":"hipster","status":200,"total_jokes":2,"total_pages":1}
```

Example: Same Value

```
{
  "current_page": 1,
  "limit": 20,
  "next_page": 1,
  "previous_page": 1,
  "results": [
    {
      "id": "GlGBIY0wAAAd",
      "joke": "How much does a hipster weigh? An instagram."
    },
    {
      "id": "xc21Lmbxcib",
      "joke": "How did the hipster burn the roof of his mouth? He ate the pizza before it was cool."
    }
  ],
  "search_term": "hipster",
  "status": 200,
  "total_jokes": 2,
  "total_pages": 1
}
```

Example: Same Value

```
{
  "current_page": 1,
  "limit": 20,
  "next_page": 1,
  "previous_page": 1,
  "results": [
    {
      "id": "GlGBIY0wAAAd",
      "joke": "How much does a hipster weigh? An instagram."
    },
    {
      "id": "xc21Lmbxcib",
      "joke": "How did the hipster burn the roof of his mouth? He ate the pizza before it was cool."
    }
  ],
  "search_term": "hipster",
  "status": 200,
  "total_jokes": 2,
  "total_pages": 1
}
```

Example

```
{"current_page":1,"limit":20,"next_page":1,"previous_page":1,"results":[{"id":"G1GBIY0wAAAd","joke":"How much does a hipster weigh? An instagram."},{id":"xc21Lmbxcib","joke":"How did the hipster burn the roof of his mouth? He ate the pizza before it was cool."}], "search_term":"hipster","status":200,"total_jokes":2,"total_pages":1}
```

JSON Syntax

- Very similar to hash literal in Ruby
 - Syntax looks identical to hash, array, number, boolean
`{"dept": "CSE", "class": 3901, "days": [1, 3, 5]}`
 - Spaces and newlines don't matter
- But there are important differences!
 - Object keys are strings (not symbols, or other types)
 - `"dept":` not `dept:`
 - Strings must be double quoted (not single)
 - `"CSE"` not `'CSE'`
 - No comments

Example: Data Access

```
{
  "current_page": 1,
  "limit": 20,
  "next_page": 1,
  "previous_page": 1,
  "results": [
    {
      "id": "GlGBIY0wAAAd",
      "joke": "How much does a hipster weigh? An instagram."
    },
    {
      "id": "xc21Lmbxcib",
      "joke": "How did the hipster burn the roof of his mouth? He ate the pizza before it was cool."
    }
  ],
  "search_term": "hipster",
  "status": 200,
  "total_jokes": 2,
  "total_pages": 1
}
```

In ruby, how do we find the id of the second joke?

(De)serialization in Ruby

□ Get JSON from an object

```
JSON.generate ([0x10, true, :age, 'hi'])
```

```
#=> "[16,true,\"age\",\"hi\"]"
```

```
JSON.generate ({mascot: "Brutus", nut: true})
```

```
#=> "{\"mascot\":\"Brutus\",\"nut\":true}"
```

□ Get an object from JSON

```
s = "{\"zip\": [43210, 43211]}"
```

```
JSON.parse(s)
```

```
#=> {'zip' => [43210, 43211]}
```

```
JSON.parse(s, symbolize_names: true)
```

```
#=> {:zip => [43210, 43211]}
```

Alternatives

- ❑ JSON is readable
 - Sometimes used for configuration files
 - ❑ VSCode: `.vscode/settings.json`
 - ❑ `.markdownlint.json`, `devcontainer.json`,...
- ❑ But JSON isn't very human-friendly
 - No comments
 - Visual clutter with lots of " marks
- ❑ Alternatives, when readability matters
 - YAML: Yet Another Markup Language
 - JSONC: JSON with comment, not standardized/universal

Web APIs

- An API contains *endpoints*, each of which is:
 - A URL path and a verb (GET or POST)
 - Required/accepted arguments
 - Returned value (often JSON)
- Roughly equivalent to a method signature
- Many ways to call an endpoint
 - Command line: curl
 - Tool: Postman, VSCode extensions rest-client,...
 - Ruby client gem: Faraday, Net::HTTP, httpx,...
 - Client library provided by the service itself: octokit for GitHub, stripe-ruby for Stripe,...

Example APIs

□ Dad Jokes

- <https://icanhazdadjoke.com/api>

□ Canvas (ie Carmen)

- <https://canvas.instructure.com/doc/api/>

□ US National Weather Service

- <https://www.weather.gov/documentation/services-web-api>

□ US Census Bureau

- <https://www.census.gov/data/developers/data-sets.html>

□ GitHub

- <https://docs.github.com/en/rest>

□ And many, many more...

- <https://github.com/public-apis/public-apis>

Demo: Calling an API

❑ Command line (curl) to find dad jokes

```
$ curl https://icanhazdadjoke.com/search?term=computer
$ curl https://icanhazdadjoke.com/search?term=computer \
  -H "Accept: application/json"
```

❑ Browser to call [Carmen API](#)

```
https://osu.instructure.com/api/v1/courses
https://osu.instructure.com/api/v1/courses?per_page=50
```

❑ Ruby gem ([HTTPX](#)) to find dad jokes

```
require 'httpx'
resp = HTTPX.get('https://icanhazdadjoke.com',
                 headers: {'accept' => 'application/json'})
puts resp.body
puts resp.json['joke']
```

API Key

- Service may require a key to use
 - Register with service, get a secret token (a long random number or string)
`8497~Xd0aaaaaIMadeThisUpzzzz`
 - Include this token in every HTTP request, eg using the Authorization header
`Authorization: Bearer 8497~Xd0aaaaaIMadeThisUpzzzz`
- Rule: never share or commit your secret token!
 - Treat it like a password
 - Dilemma: Your code needs to use it, so it needs to be stored somewhere...

One Strategy: Environment Variables

- ❑ **First:** Keep `.env` file out of commits!

```
# .gitignore
.env
```

- ❑ **Then:** Create the `.env` file for secret(s)

```
# .env
CANVAS_TOKEN=8497~Xd0aaaaaIMadeThisUpzzzz
```

- ❑ **Helpful:** Create a *sample* `.env` file with dummy value(s)

```
# .env.template
CANVAS_TOKEN=CANVAS_TOKEN_SECRET
```

- ❑ **Use environment variable(s) in client code**

```
require 'dotenv'      # useful gem for reading .env files
Dotenv.load           # creates ENV hash of values in .env
auth = "Bearer #{ENV['CANVAS_TOKEN']}"
req.header['Authorization'] = auth      # set Auth header
```

Getting an API Key

- ❑ GitHub
 - Login, Settings > Developer Settings
 - Personal access tokens > Tokens
- ❑ Canvas
 - Login, Account > Settings
 - Approved Integrations, "+ New Access Token"
- ❑ Use a meaningful name for token
- ❑ Value typically shown just one time

Summary

- Passing arguments
 - GET: query string (url-encoded)
 - POST: body (several different encodings)
- JSON
 - Syntax for describing values
 - Just a few basic types (object, array, text, number...)
 - Useful for (de)serialization, while also human-readable
- API endpoints
 - Response body is often JSON
- API keys
 - Protect secrets, eg with private .env file
 - Use in request header to legitimize source