Working with Web APIs

Lecture 13

Passing arguments: GET

- Arguments are key-value pairs

```
Mascot: Brutus Buckeye
Dept: CS&E
```

- Can be encoded as part of URL

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

- application/x-www-form-urlencoded
  - Each key-value pair separated by & (or ;)
  - Each key separated from value by =
  - Replace spaces with + (arcane!)
  - Then normal URL encoding

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

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: POST

- Encoded as part of the request *body*
- Advantages:
  - Arbitrary length (URLs are limited)
  - Arguments not saved in browser history
  - Result not cached by browser
  - Slightly more secure (not really)
    - Args are less likely to be accidentally shared, because they aren't obvious in the location bar
- Content-Type indicates encoding
  - application/x-www-form-urlencoded
    - Same encoding as used with GET
  - multipart/form-data
    - Better for binary data (else 1 byte→3 bytes)
  - More options too:
    - application/xml, application/json, …

Passing Args: 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 Args: Summary

- Arguments in GET requests
  - Request query string
  - Limited length, highly visible
  - application/x-www-form-urlencoded
- Arguments in POST requests
  - Request body
  - No size limit, not bookmarked
  - Choices for how to encode, most common:
    - 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: requests and responses
- MIME type: application/json

JSON 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"]`

- Object: set of name/value pairs, {...}

  ```
  {"mascot": "Brutus", "age": 19, "nut": true}
  ```

Example

{"current_page":1,"limit":20,"next_page":1,"previous_page":1,"results":[{"id":"GlGBIY0wAAd","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": "GlGBIY0wAAd",
      "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
}
```

Syntax

- Very similar to hash literal in Ruby
  - Inspired by object literal in JavaScript

```
{"dept": "CSE", "class": 3901}
```

  - Spaces and newlines don't matter
- But not identical!
- Important differences
  - Keys are strings (not symbols)
    - `"dept":` not `dept:`
  - Strings are double quoted (not single)
    - `"CSE"` not `'CSE'`
  - No comments

Example


```
{
  "current_page": 1,
  "limit": 20,
  "next_page": 1,
  "previous_page": 1,
  "results": [
    {
      "id": "GlGBIY0wAAd",
      "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
}
```


```
x['results'][1]['id'] #=> 'xc21Lmbxcib'
```

(De)serialization in Ruby

- Get JSON from an object

```
JSON.generate ([0x10, true, :age, 'hi'])
#=> "[16,true,\"age\",\"hi\"]"
```

- Get an object from JSON

```
s = "{\"zips\": [43210, 43211]}"
JSON.parse(s)
#=> {'zips' => [43210, 43211]}
JSON.parse(s, {symbolize_names: true})
#=> {:zips => [43210, 43211]}
```

Alternatives

- JSON is readable
  - Sometimes used for configuration files
    - VSCode: .vscode/settings.json
    - .markdownlint.json, devcontainer.json,…
- But JSON isn't human-friendly
  - No comments
  - Visual clutter with lots of " marks
- Alternatives, when readability matters
  - YAML: yet another markup language
  - JSONC: adds comment, not universal

Web APIs

- API contains endpoints, each of which:
  - verb (GET or POST) and URL path
  - Accepted arguments
  - Returned value (typically JSON)
- Roughly equivalent to a method signature
- Many ways to call an endpoint
  - Command line: curl
  - Tool: VSCode extension rest-client, Postman
  - HTTP client library: (Faraday, Net::HTTP)
  - 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

API Key

- Service may require a key to use
  - Register with service, get a secret token (ie a long random number or string)
  - Include this token in every HTTP request, eg using the Authorization header

    `Authorization: Bearer canvas_12341234aaaaffff`
- Golden 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…

Solution Strategy: Env Variable

- Create .env file for secrets

```
# .env
CANVAS_TOKEN=YOURSECRETVALUE
```

- Keep .env out of commits!

```
# .gitignore
.env
```

- Create sample with dummy value

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

- Use environment variable in client code

```
require 'dotenv'
Dotenv.load # looks for .env file
auth = "Bearer #{ENV['CANVAS_TOKEN']}"
req.header['Authorization'] = auth
```

Getting an API Key

- GitHub
  - Login, Settings > Developer Settings
  - Personal access tokens > Tokens
- Canvas
  - Login, Account > Settings
  - Under "Approved Integrations", "+ New Access Token"

- Use 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