LESSON 6: ANATOMY OF REQUEST/RESPONSE AND POST

FETCH UP UNTIL NOW

- Before this class we have used fetch in a somewhat transparent fashion: we give it a URL, some data comes back, and we process it.
- This is fine for simple GET requests, but once we build up more complex requests or need to expect/handle a variety of responses we need to break down each part more thoroughly.

THE REQUEST

- This is the first part of fetch
  - fetch('http://example.com/api/photos?apikey=a3vkej', { options })
- This will cause the browser to send a lot of information:
  - URL
  - HTTP method
  - Query params
  - "URL Params"
  - Request Body
  - HTTP headers
    - Content Negotiation
    - Auth

URL

- ‘http://example.com/api/photos'
- Three parts:
  - The protocol: http:// (HyperText Transfer Protocol)
  - The base url: example.com
  - The path (or resource path): /api/photos

HTTP METHOD

- GET - Request data
- POST - Add data
- PUT - Update data
- DELETE - Delete data
- The same path can support multiple methods:
  - /api/photos (can ask for a photo with GET, add a new one with POST)

QUERY PARAMETERS

- The part of the URL after the path that contains extra information that doesn't fit sensibly into the path
- The first query param must come after the ?, which is placed after the end of the path.
- Subsequent query params are separated by &
- Format is <nameOfParam>=<paramValue>
- example.com/api/photos?apiKey=ae3dd&feature=popular

URL PARAMS

- URL params are not true params, but they come up often in REST APIs for identifying a particular resource. They are usually an ID, and are denoted by ':' in the API documentation.
- If the API documentation states a path is

  /photos/:photoId/comments

  it is saying that the photoId section of the path should be filled with the particular photoId you are trying to retrieve comments for.

  - photos/12277/comments will return comments for photo with id 12277.

BODY

- For our purposes, the request body will only be used with the POST method, and will usually contain JSON data. It can also store text data, or form encoded data.
- When POSTing to a particular endpoint, say to add a comment to a photo, the body is the JSON object that contains the comment data.
- With fetch it is specified in options.

BODY EXAMPLE

- Option Object:
  { body: JSON.stringify({ text: “This is a comment!’}) }
- JSON.stringfy the body because we can only actually send text. The server will parse it back into an object it understands.

HEADERS

- HTTP headers are extra data that get sent with the request which provide information about the request itself.
- There are many headers, and you can even define your own. For many cases we won't have to do this.
- Three common headers are:
  - Content-Type: Tells the server what kind of data you're sending (application/json, for example).
  - Auth: A simple username/password combination can be sent in a Basic-Auth header.

POST

- For most POST endpoints that are expecting JSON data, we have to manually set the Content-Type to ‘application/json’ which tells the server we are sending JSON data.
- If we don’t send this we will get a Bad Request error.
- We do this with fetch by using the headers property in options:
- { headers: { ‘Content-Type’: ‘application/json’ } }

THE RESPONSE

- Response objects contain much of the same information we send to the server in the request:
  - Headers
  - Body
- The main difference is Response objects have a status code associated with them as well, which indicate how the server responded.

STATUS CODES

- They come in five groups, each prefixed with a number
  - 1xx - Informational
  - 2xx - Success
  - 3xx - Redirect
  - 4xx - Client Error
  - 5xx - Server error

COMMON CODES

- 200 - OK, everything worked fine
- 301 - URL move permanently, use the new one
- 400 - Bad Request, you messed up formatting the request
- 401 - Unauthorized - I don't know who you are, authenticate
- 403 - Forbidden - I know who you are but you're not allowed to do this
- 404 - Not found - The URL you tried to request doesn't exist
- 405 - Method not allowed - The HTTP method isn't allowed for this path…you tried a POST for a path that only accepts GET, for example
- 500 - The server messed up somehow
- 503 - Server is busy and can't handle the request right now

DETAILED CODES

- All codes are here https://developer.mozilla.org/en-US/docs/Web/HTTP/Status but the ones last side are the ones you will typically encounter.

FETCH AND ERROR HANDLING

- So far we have done .catch to handle fetch errors, but unfortunately fetch works in a strange way.
- If there is not a network error but you receive an error code, such as 400, fetch will NOT count that as an error and the catch block will not run.
- Instead in the first .then block we can check the status prop on the response object
- It also has a shorthand for checking if that status === 200, response.ok will be true or false

GETTING THE DATA ON A SUCCESSFUL REQUEST

- We usually know what return type to expect from the API documentation
- If not, we can check the response.headers object and search for the Content-Type header.
- Once we get the response object, we can call a number of methods to parse the body of the response, such as .json() or .text()