REST

Overview

- Stands for **RE**presentational **S**tate **T**ransfer
- An architectural style, not a standard or API
- Described in Roy Fielding's dissertation in 2000 (chapter 5)
  - http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- Main ideas
  - a software component requests a "**resource**" from a service
    - by supplying a resource identifier and a desired media type
    - resource identifier can be a URL and request can be made using Ajax
  - a "**representation**" of the resource is returned
    - a sequence of bytes and metadata to describe it (can be JSON; name/value pairs in HTTP headers, ...)
    - can contain identifiers of other resources
  - obtaining this representation causes the software component to "**transfer**" to a new "**state**"

Typical REST

- Relies on HTTP
- Uses carefully selected URLs to identify resources
- Uses HTTP methods for specific operations

a.k.a. HTTP verbs

primarily for Create-Retrieve-Update-Delete (**CRUD**) operations

  - **GET** - retrieves representation of existing resource identified by URL
  - **POST** - creates a new resource using data in body
    - URL of new resource is not yet known
    - often the URL of the request identifies an existing parent resource
    - return URL of new resource in **Location header** and **status 201 Created**

ex. POST to URL for a music artist, supplying data describing a new CD; URL of new CD resource is returned

  - **PUT** - updates existing resource identified by URL using data in body
    - URL of resource is already known
    - like POST, body contains entire description of resource
    - can also create a new resource if its URL is already known
  - **PATCH** - like PUT, but body only describes changes
    - other attributes/properties retain their current value
  - **DELETE** - deletes existing resource identified by URL
  - **POST** - for everything else (non-CRUD operations)

Idempotent

- Means doing something multiple times is no different than doing it once
  - in REST this applies to server-side state and responses
- GET, PUT, and DELETE requests should be idempotent
  - it’s possible for data returned by repeated GET requests to differ,
    but the differences shouldn’t be caused by processing the GET request
- POST requests are not necessarily idempotent
  - each request can cause different changes to server state
    based on the state at the time the request is processed
  - POST examples
    - can be used to create a new resource and each could be assigned a different identifier
    - can be used to order an item; remaining inventory changes, so subsequent requests can fail
- These rules aren’t enforced,
  but violating them goes against the HTTP specification

Ajax

- REST calls are made from web clients using Ajax
- Can use low-level XMLHttpRequest approach
- Typically a library is used instead
- Many choices
  - Fetch, axios, superagent, jQuery, ...
- We’ll explore the two most popular, Fetch and axios

Fetch API

- "The Fetch Standard defines the `fetch()` JavaScript API, which exposes ... networking functionality at a fairly low level of abstraction."
- A Web Hypertext Application Technology Working Group (WHATWG) standard
  - https://fetch.spec.whatwg.org/
- Browser support
  - Chrome 42+, Firefox 39+, not in IE, not in Safari
- Polyfill
  - https://github.com/github/fetch
  - works in Chrome, Firefox, IE9+, and Safari 6.1+
  - **`npm install --save whatwg-fetch`**
  - to make `fetch` function available globally,
    `import 'whatwg-fetch';`
  - HTTP method names can be lowercase, except **`PATCH`**!
    - just make them all uppercase to avoid confusion

axios

- "Promise based HTTP client for the browser and Node.js"
- https://github.com/mzabriskie/axios
- From Matt Zabriskie
- Less verbose than Fetch API
- To use
  - `npm install --save axios`
  - `const axios = require('axios');`

GET

- To send a GET request to obtain text

```
try {
  const res = await fetch(restUrl);
  const text = await res.text();
  // Do something with text.
} catch (e) {
  // Handle error.
}
```

```
try {
  const res = await axios.get(restUrl);
  const text = res.data;
  // Do something with text.
} catch (e) {
  // Handle error.
}
```

- To send a GET request to obtain JSON

```
try {
  const res = await fetch(restUrl);
  const obj = await res.json();
  // Do something with obj.
} catch (e) {
  // Handle error.
}
```

```
try {
  const res = axios.get(restUrl);
  const obj = res.data;
  // Do something with obj.
} catch (e) {
  // Handle error.
}
```

automatically parses JSON when response `Content-Type` is `application/json`

DELETE

- To send a DELETE request

```
try {
  await fetch(restUrl, {method: 'DELETE'});
  // Do something after
  // successful delete.
} catch (e) {
  // Handle error.
}
```

```
try {
  await axios.delete(restUrl);
  // Do something after
  // successful delete.
} catch (e) {
  // Handle error.
}
```

POST/PUT/PATCH Text

- To send a POST (or PUT or PATCH) request with a text body

```
try {
  const res = await fetch(restUrl, {
    method: 'POST',
    headers:{'Content-Type': 'text/plain'},
    body: text
  });
  // Do something with success response.
} catch (e) {
  // Handle error.
}
```

```
try {
  const res = await axios.post(restUrl, text, {
    headers: {'Content-Type': 'text/plain'}
  });
  // Do something with success response.
} catch (e) {
  // Handle error.
}
```

POST/PUT/PATCH JSON

- To send a POST (or PUT or PATCH) request with a JSON body

```
try {
  const res = await fetch(restUrl, {
    method: 'POST',
    headers:{'Content-Type': 'application/json'},
    body: JSON.stringify(obj)
  });
  // Do something with success response.
} catch (e) {
  // Handle error.
}
```

can also add **`Accept`** header to describe acceptable response MIME types

```
try {
  const res = await axios.post(restUrl, obj, {
    headers: {'Content-Type': 'application/json'}
  });
  // Do something with success response.
} catch (e) {
  // Handle error.
}
```

can also send a POST with no body

no need to stringify the object being sent

Fetch Caveats

- From polyfill documentation
  - "The Promise returned from `fetch()`
    won't reject on HTTP error status
    even **if** the response is a HTTP **404 or 500**.
    Instead, it **will resolve normally**,
    and it **will only reject on network failure**,
    or if anything prevented the request from completing."
  - "**By default**, fetch **won't send any cookies to the server**,
    resulting in unauthenticated requests
    if the site relies on maintaining a user session."
    - "**To automatically send cookies** for the current domain,
      the credentials option must be provided"

check **`res.status`**
or **`res.ok`**
in success handler

fetch-util.js

```
const options = {};

export async function getJson(url) {
  const res = await fetch(url, options);
  return res.json();
}

export async function getText(url) {
  const res = await fetch(url, options);
  return res.text();
}

export async function postJson(url, obj) {
  const body = JSON.stringify(obj);
  const headers = {'Content-Type': 'application/json'};
  const res = await fetch(url,
    {...options, method: 'POST', headers, body});
  return res;
}
```

call these functions inside a `try/catch`