Fetch

Browser XHR (XMLHttpRequest) for service calls

- It was horrible
- Many libraries made to help (jquery, axios, etc)

Now we have `fetch()`!

- Native to all modern browsers
- Friendly, promise-based API
- No need for those other libraries

Fetch is called with a url

- Can be Fully Qualified URL
- Can be absolute path
- Can be relative path

```
fetch('http://example.com/cats');
fetch('/cats');
fetch('cats');
```

If calling a service on same server

- Use absolute path: `fetch('/cats');`

If calling a service on a different server

- Use fully qualified: `fetch('http://example.com/cats');`

Fetch returns a promise

```
const promise = fetch('/cats');
promise.then( () => console.log('fetch complete') );
```

The promise resolves with a Response object

- MDN Response
- **https://developer.mozilla.org/en-US/docs/Web/API/Response#instance_properties**

```
fetch('/cats')
  .then( response => console.log(response.status) );
```

Our example /cats endpoint

Assume `/cats` endpoint returns a JSON array of strings

```
[
  "Jorts",
  "Jean",
  "Maru"
]
```

Response object does NOT have the parsed body

You need the data from the request body

- Body not yet parsed
- Body may not be fully received yet

Call a method to parse the body

- `.text()` or `.json()` (examples)

These parsing methods **are themselves async**

- Return promises

```
fetch('/cats')
  .then( response => response.json() ) // Parses JSON to JS
  .then( body => console.log(body) ); // body is JS
```

Example of parsing

```
fetch('/cats')
  .then( response => response.json() )
  .then( body => console.log(body) );
console.log(1);
```

- `response` is the response object
    - Does NOT have the JSON string in the body
- `response.json()` returns a promise
    - Promise resolves when JSON body is parsed
    - Resolves with the parsed JS (not JSON!)
    - JSON is a text string, our JS is an array
- `body` here is array: `[ "Jorts", "Jean", "Maru" ]`
- `console.log(1);` happens BEFORE the parsing

Using the body

```
<ul class="example"></ul>
```

```
const list = document.querySelector('.example');
fetch('/cats')
  .then( response => response.json() )
  .then( cats => {
    const names = cats.map(
      name => `<li>${name}</li>`
    ).join('')
    list.innerHTML = names;
  });
```

But: This updates the DOM directly - bad idea!

What is better?

Better design

```
<ul class="example"></ul>
```

```
import state, {updateNames} from './state';

const listEl = document.querySelector('.example');

fetch('/cats')
.then( response => response.json() )
.then( cats => {
  updateNames(cats); // update state
  render();
});

function render() {
    list.innerHTML = state.names.map(
      name => `<li>${name}</li>`
    ).join('')
}
```

More improvements to come!

Why better?

state is maintained in variables

- Not just in the current DOM
- Can rebuild DOM at any time from state
- Can consult state without checking DOM
- Keeps state management simple
    - Unimpacted by changes to the DOM

We can change state in many places

- Always call `render()` or `renderSomeSection()`
    - "render" is my name, but common concept

Handling errors

`fetch` promises reject if communication fails

- Network errors reaching server
- **Does NOT reject on service error message**

Service errors are successful communication

- `fetch()` promise will **resolve**, not **reject**
- Only network errors will be caught by `catch()`

How to detect and respond to service error messages?

Service Errors

- When you can REACH the service
- But something is wrong
- Maybe you sent bad data
- Maybe server has unrelated problem

How does service tell calling program?

Errors by Status code

Some services return *meaningful* HTTP Status codes

- Like REST services (more later)
- **2xx** status codes === Success
    - Not an error
- **3xx** === Redirection (not really errors?)
    - Rare for service calls
- **4xx** === Client Caused Errors
- **5xx** === Server Errors
- May be more detail in the body
    - Body may have its own structure (JSON?)

Detecting Status Codes Indicating Errors

For these we can check for the HTTP status code

- Services with good status codes are important
- `response.ok` for "good" status code ranges

This only applies if HTTP status codes are meaningful!

- Some services may return **200 OK** even for errors!

Errors by Content

Some services don't use *meaningful* HTTP Statuses

- Instead send error indicator in the body data

You will have to parse the body then examine it

Fetch Promise rejects if network error

To check for connection error

- Catch before parsing response
- Rethrow/reject with more info

```
fetch('/cats')
  .catch( () => {
    return Promise.reject( {
      error: 'network-error'
    });
  })
  .then( response => {
    // not run in case of network error
```

You decide what your error case looks like

Error Detection Breakdown

```
fetch('/cats')
    .catch( () => { // network error caught here
      // rethrow/reject with your own formatted value
    })
    .then( response => { // Just response status so far!
      if(response.ok) {
        // return a parsing method call like response.json()
      }
        // If service gives meaningful status
        // - throw/reject with a formatted value
        // - may need to parse error response body
        //     - and throw/reject that
        // If service does not give meaningful status
        // - something went wrong (like 404)
        // - throw/reject with a formatted value
        // - error response body unlikely to help much
    })
    .then( cats => { // parsed response body
      // Do we need to check it for error indicator
      // - and throw/reject?
    }) // ...
```

Error example

```
<ul class="example"></ul>
<div class="status"></div>
```

```
const status = document.querySelector('.status');
fetch('/cats')
    .catch( () => Promise.reject({ error: 'network' }) );
    .then( response => {
      if(response.ok) { return response.json(); }
      // This example service sends JSON error bodies
      return response.json().then(
        err => Promise.reject(err)
      ); // returns rejected promise
    })
    .then( cats => {
      updateNames(cats); // method to update state
      render();
    })
    .catch( err => status.innerText = err.error );
```

Final `.catch()` gets both network and service errors

Reporting Errors to the User

You need to tell the user

- If they need to take action
- Or need to know info is out of date

`console.log()` is NOT telling the user

- Did you look there before this class?

Putting Errors in HTML

- User needs to see messages on the page
    - Messages need to be in HTML
    - = Error indicator has be in state
- User needs to see a FRIENDLY message
    - What should they do?

Error Indicator in state, not Error Message!

Often DON'T want to show the message from server

- i18n/l12n issues
- Service rarely User-friendly language
- Service may have many clients - can't change

Message text usually better in View, not Model

- Store indicator in Model/state
- View/Render translates indicator to specific text
- Changing text requires NO change to Model/state

Translating Error Messages

Service may report an error code

- Varies by service author
- Front end code "translates" to user friendly

```
const MESSAGES = {
  'network-error': "Server unavailable, please try again",
  'invalid-name': "Name not found, please correct",
  default: "Something went wrong, please try again",
};
// ...
  .catch( error => { // If 'error' is the code
    const message = MESSAGES[error] || MESSAGES.default;
    // Simple example, translation better done in View/Render
    // ...
```

Manually Testing Errors

Easy to test errors where you send bad data

- But how to test server unavailable?

Two options

- Stop server and try front end service call
- DevTools - Network
    - "No throttling" to "Offline"
    - Remember to change back after test!

Error Tips

- Don't leave the user confused
- console.log() is **NOT** error handling
- You rarely SHOW the exact service error message

Students lose points on assignments and projects

- Every single semester
- No matter how I beg and plead
- Please break the trend

Tell the user what they need to do

- Just like you see on websites!

Different HTTP methods

`fetch()` defaults to GET method

It accepts an optional object

- The `method` key allows you to set the method

```
fetch('/cats', {
  method: 'POST'
})
```

More HTTP Methods

`fetch()` supports more methods than GET and POST

- DELETE
- PUT
- PATCH
- OPTIONS, TRACE, and HEAD
    - rarely called in fetch()

More discussion later

- For now: they are all called by setting `method`

Sending Data

Query params are sent as part of the URL

- the first argument to `fetch()`

Body params can be sent as the `body` option

- Remember: Not with GET
- Body params can be in multiple formats

```
// Not yet complete
fetch('/cats', {
  method: 'POST',
  body: JSON.stringify({ name: 'Maru', age: 12 }),
})
```

Converting Data to a JSON Body

- HTML Forms use **url-encoded** by default
  - `key=value&key=value&key=value`
- We will instead send our data in the body as **JSON**
  - Text string that LOOKS like JS
  - Handles more complex data

```
const cat = {
  name: "Jorts",
  age: 3,
  toys: [
    'pipe cleaner',
  ],
};
const json = JSON.stringify(cat); // convert to JSON
console.log(json);
// '{"name":"Jorts","age":3,"toys":["pipe cleaner"]}'
```

Telling Server about Encoding of body data

Body of request sent to server

- Can be encoded in different ways
- How does server know how we sent it?
- We tell it by setting a header in the request!
    - Header: data about request
    - Body: data IN request

Sending Headers

There is a `headers` property

- Adds to/overrides default headers
- Need to tell server what format body is in

```
fetch('/cats', {
  method: 'POST',
  headers: {
    'content-type': 'application/json'
  },
  body: JSON.stringify({ name: 'Maru', age: 12 }),
})
```

Set `content-type` header when formatted body!

Many servers will not parse the body otherwise

- Confusing error messages about missing data

What about cookies on service call web requests?

Cookies and `Auth` headers

- Controlled by the `credentials` option to `fetch()`
- `omit`, `same-origin` (default), `include`
    - "origin" is protocol+domain+port
    - Compares fetched url to url of current page
- Controls sending cookies
    - And setting received cookies

```
fetch('/cats', {
  method: 'GET',
  credentials: 'include',
})
```

Separating Concerns

So far

- `fetch()`
- `.then()/.catch()` chain
- Update state
- Call render

But we have excessive coupling!

- Our call to `fetch()`
- How we use the data
    - Update state
    - Render

Returning the promise

```
function fetchCats() {
 return fetch('/cats')
  .catch( () => Promise.reject({ error: 'network' }) );
  .then( response => {
    if(response.ok) { return response.json(); }
    // This example service sends JSON error bodies
    return response.json().then(err => Promise.reject(err) );
  });
}
```

- Makes call
- Converts body/error
- Does NOT alter state or DOM
- Returns the promise

Using the Promise

The **caller** of the function that returns the promise

- Can attach further callbacks
    - To use results
        - Update state
        - `render()`
- Making call and using results
    - Now **decoupled** (concerns separated!)
- That fetching function reusable

Separated fetching concern example

```
// fetchCats() returns promise that resolves with data
// or rejects with an error object
fetchCats() // actual fetch code abstracted away
  .then( cats => {
    state.names = cats;
    render();
  })
  .catch( err => {
    state.error = err?.error || "Unknown Error";
    render();
  });
```

Remember, in `render()`, something like:

```
const errorText = MESSAGES[state.error] || MESSAGES.default;
```